TANDEM 16

NonStop™
SYSTEMS

T/TAL
LANGUAGE
PROGRAMMING MANUAL

T16/8013 A03
82013

INTRODUCTION TO PROGRAMMING THE TANDEM 16

AND

TANDEM/TRANSACTION APPLICATION LANGUAGE

PROGRAMMING MANUAL

TANDEM COMPUTERS INCORPORATED
19333 Vallco Parkway
Cupertino, California 95014

Product No. T16/8013 A03
Part No. 82013

November 1978
Printed in U.S.A.

NOTICE

The following are trademarks of Tandem Computers Incorporated, and may be used only to describe products of Tandem Computers Incorporated.

ENFORM	EXPAND	NonStop
ENSCRIBE	EXTEND	TANDEM
ENVOY	GUARDIAN	TGAL

Pages	Effective Date	Software Product and Version
Title	Nov. 1978	
List of Effective Pages	Nov. 1978	
Preface	Nov. 1978	
Reader's Guide	Mar. 1978	
(1)-1	Feb. 1977	
1-1 thru 1-2	Feb. 1977	
1-3 thru 1-4	Mar. 1978	
1-5 thru 1-12	Feb. 1977	
(2)-1	Mar. 1978	
(2)-2	Feb. 1977	
(2)-3	Mar. 1978	
(2)-4	Nov. 1978	
(2)-5	Mar. 1978	
(2)-6	Aug. 1978	
2.1-1 thru 2.1-2	Mar. 1978	T9200B08
2.1-3 thru 2.1-7	Feb. 1977	T9200B08
2.1-8 thru 2.1-9	Mar. 1978	T9200B08
2.1-10 thru 2.1-21	Feb. 1977	T9200B08
2.2-1 thru 2.2-5	Feb. 1977	T9200B08
2.3-1	Mar. 1978	T9200B08
2.3-2 thru 2.3-4	Feb. 1977	T9200B08
2.4-1 thru 2.4-6	Feb. 1977	T9200B08
2.5-1 thru 2.5-4	Feb. 1977	T9200B08
2.6-1 thru 2.6-2	Feb. 1977	T9200B08
2.6-3	Mar. 1978	T9200B08
2.7-1 thru 2.7-5	Mar. 1978	T9200B08
2.8-1 thru 2.8-39	Feb. 1977	T9200B08
2.9-1	Mar 1978	T9200B08
2.9-2	Feb. 1977	T9200B08
2.10-1 thru 2.10-5	Mar. 1978	T9200B08
2.11-1 thru 2.11-6	Feb. 1977	T9200B08
2.11-7	Mar. 1978	T9200B08
2.11-8 thru 2.11-20	Feb. 1977	T9200B08
2.12-1 thru 2.12-8	Feb. 1977	T9200B08
2.13-1 thru 2.13-2	Feb. 1977	T9200B08
2.14-1 thru 2.14-7	Feb. 1977	T9200B08
2.15-1 thru 2.15-22	Feb. 1977	T9200B08
2.15-23 thru 2.15-26	Mar. 1978	T9200B08
2.16-1 thru 2.16-17	Feb. 1977	T9200B08
2.17-1 thru 2.17-15	Feb. 1977	T9200B08
2.17-16	Mar. 1978	T9200B08
2.17-17 thru 2.17-19	Feb. 1977	T9200B08
2.17-20 thru 2.17-21	Mar. 1978	T9200B08
2.17-22 thru 2.17-32	Feb. 1977	T9200B08
2.18-1 thru 2.18-14	Feb. 1977	T9200B08
2.19-1 thru 2.19-9	Mar. 1978	T9200B08
2.20-1 thru 2.20-2	Mar. 1978	T9200B08
2.20-3	Nov. 1978	T9200C01
2.21-1 thru 2.21-8	Feb. 1977	T9200B08
2.22-1 thru 2.22-15	Feb. 1977	T9200B08
2.22-16	Mar. 1978	T9200B08

2.22-17 thru 2.22-33	Feb. 1977	T9200B08
2.23-1 thru 2.23-5	Mar. 1978	T9200B08
2.23-6	Nov. 1978	T9200B08
2.23-7 thru 2.23-10	Mar. 1978	T9200B08
2.23-11	Nov. 1978	T9200B08
2.23-12 thru 2.23-26	Mar. 1978	T9200B08
2.24-1 thru 2.24-6	Aug. 1978	T9200C00
(app)-1	Nov. 1978	
A-1 thru A-8	Feb. 1977	T9200B08
B-1 thru B-11	Mar. 1978	T9200B08
C-1 thru C-4	Feb. 1977	
D-1 thru D-15	Nov. 1978	T9200C01
Index-1 thru Index-15	Nov. 1978	

MANUAL VERSION A00. DATE: FEBRUARY 15, 1977
MANUAL VERSION A01. DATE: MARCH 15, 1978
MANUAL VERSION A02. DATE: AUGUST 15, 1978
MANUAL VERSION A03. DATE: NOVEMBER 30, 1978

Note: This manual will be updated as necessary to correct errors in the manual or to reflect changes or additions to the software. The changes to the manual will be made available in the form of individual changed pages in a "manual update package". A subscription service is available for our customers who would like to receive manual update packages automatically.

The programming manual is organized into two sections followed by the appendices and an index. The sections are:

Section 1. Introduction to Programming the Tandem 16
- Program Characteristics
- T/TAL
- GUARDIAN
- Program Development
- Conventions in this Manual

Section 2. T/TAL Language
- Language Characteristics
- Program Organization
- Data Declarations
- Statements
- Compiler Commands
- Advanced Features
- Structures

Generally, the information in section two falls into either of two categories. The first (and largest) part of the section contains information requiring that the programmer have a good understanding of the overall system but a less than thorough knowledge of specific hardware details. The second part of the section deals with so-called advanced features and requires a knowledge of the Tandem 16 hardware registers, machine instructions, and/or operating modes.

The four appendices contain the following information:

Appendix A. T/TAL Language Summary

B. BNF Syntax for T/TAL

C. ASCII Character Set

D. T/TAL Compiler Diagnostic Messages

For more information regarding the Tandem 16 Computer System, refer to the following manuals:

* Tandem 16 System Description [Product no. T16/8000]

- Overview of the Tandem 16 system, hardware and software
- Details of the hardware from a programming standpoint
- Tandem 16 approach to NonStop programming
- Tandem 16 machine instructions
- Description of the operating system and its responsibilities

* Tandem 16 Operating Manual [Product no. T16/8019]

- Interactive Command Interpreter (COMINT)
- Interactive Text Editor (EDIT)
- File Utility Program (FUP)
- Backing up/restoring disc files (BACKUP/RESTORE)

- Peripheral Utility Program (PUP)
- Program File Editor (UPDATE)
- System Generation (SYSGEN)
- Console error messages
- System Load Procedures
- Peripheral User's Guide

The following is a list of computer system functions and applicable manuals. The manuals are

T16/8000 SYSTEM DESCRIPTION MANUAL

T16/8008 SORT OPERATING AND PROGRAMMING MANUAL

T16/8013 T/TAL PROGRAMMING MANUAL

T16/8014 GUARDIAN PROGRAMMING MANUAL

T16/8015 GENERAL PURPOSE PROCEDURES PROGRAMMING MANUAL

T16/8017 ENSCRIBE PROGRAMMING MANUAL

T16/8018 ENVOY PROGRAMMING MANUAL

T16/8019 OPERATING MANUAL

To obtain a comprehensive understanding of the Tandem 16 Computer System, read the first six items in the given order.

If it is desired to:	Read:
acquire general information about the system	T16/8000, INTRODUCTION
acquire general information about programming the system	T16/8013, INTRODUCTION
acquire general information about operating system services	T16/8014, INTRODUCTION
acquire general information about the human interface to the system	T16/8019, INTRODUCTION, COMINT
acquire specific knowledge about the T/TAL language	T16/8013, TAL
programmatically create, read, write, and purge unstructured disc files, communicate with terminals, printers, magnetic tapes, and programs	T16/8014, FILE SYSTEM
programmatically create, read, write and purge key-sequenced, relative, or entry-sequenced files	T16/8017, ENSCRIBE
communicate over data communication lines	T16/8018, ENVOY
create and modify source language programs	T16/8019, EDIT
programmatically run and stop programs, change execution priority, etc.	T16/8014, PROCESS CONTROL

perform utility operations such as number conversion, requesting the time-of-day, requesting the file name of the home terminal, etc.	T16/8014, UTILITY PROCEDURES
acquire specific knowledge about the interface between the Tandem-supplied Command Interpreter program and application programs	T16/8013, COMINT/ APPLICATION INTERFACE
read EDIT-format disc files	T16/8015, GENERAL PURPOSE PROCEDURES
perform general-purpose file error recovery	T16/8015, GENERAL PURPOSE PROCEDURES
compile a source program	T16/8019, COMINT, TAL/XREF
run an object program	T16/8019, COMINT
debug a program	T16/8014, DEBUG
write a NonStop program using the Checkpointing Facility	T16/8014, CHECKPOINTING FACILITY
display disc file information, rename, and purge disc files	T16/8019, COMINT, FUP
create disc files and display detailed disc file information	T16/8019, FUP
alter disc file characteristics	T16/8019, FUP
load data into structured disc files	T16/8019, FUP
sort data records by commanding SORT through the Command Interpreter	T16/8008, SORT
sort data records by commanding SORT programmatically	T16,8008, SORT
sort data in memory	T16/8015, GENERAL PURPOSE PROCEDURES
display system status information (see what programs are running and where)	T16/8019, COMINT
make a backup copy of one or more disc files on magnetic tape	T16/8019, BACKUP
restore one or more disc files from a backup copy on magnetic tape	T16/8019, RESTORE

make a copy of a file (i.e., disc to disc, disc to tape, tape to disc, etc.)	T16/8019, FUP
display all or part of the contents of a file	T16/8019, FUP
introduce a new disc pack into the system by formatting or labeling the pack, mount a previously labeled pack	T16/8019, PUP
cause further open requests to a file to be rejected	T16/8019, PUP
list spare tracks, bad tracks, and free space on a disc volume	T16/8019, PUP
copy all files from one volume to another volume	T16/8019, PUP
revive the failed device of a mirror volume by copying the image on the up device to it	T16/8019, PUP
list characteristics of i/o devices	T16/8019, PUP
specify the device where console messages are to be listed	T16/8019, PUP
acquire specific knowledge about the Tandem 16 hardware	T16/8000, HARDWARE
acquire specific knowledge about the Tandem 16 operating system	T16/8000, OPERATING SYSTEM
make the alternate path to a device become the primary path	T16/8019, PUP
remove an interprocessor bus from system use, reinstate an interprocessor bus for system use	T16/8019, COMINT
remove (i.e., down) a device from system use. Reinstate a "downed" device for system use.	T16/8019, PUP
select procedures from existing object program files to create a new object file	T16/8019, UPDATE
display or modify an object program file	T16/8019, UPDATE
generate a configured operating system, run the SYSGEN program	T16/8019, SYSGEN

acquire a definition of console messages and processor halts	T16/8019, MESSAGES
backup all files in the system on a regular basis and restore files in the event files are lost	T16/8019, SYSTEM BACKUP/RESTORE PROCEDURES
put a newly-installed system into operation, install a new operating system, cold load a system	T16/8019, SYSTEM LOAD PROCEDURES
use and maintain system peripheral devices	T16/8019, PERIPHERAL USER'S GUIDE
install user-written procedures into the operating system library code area	T16/8019, SYSGEN
install user-written i/o drivers or i/o processes into the system process code area	T16/8019, SYSGEN
program the Tandem 16 at an assembly language level	T16/8000, INSTRUCTION SET; T16/8013, TAL

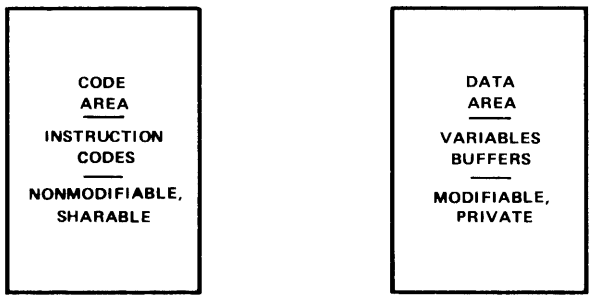
INTRODUCTION TO PROGRAMMING THE TANDEM 16

PROGRAM CHARACTERISTICS

The basic unit of information in the Tandem 16 is the 16-bit word. The word defines the Tandem 16's machine instruction length and its logical addressing range.

A Tandem 16 program executing in a processor module consists of 1) a code area that contains the executable machine instruction codes and 2) a separate data area that contains the program's variables and buffers. The code area for a given program consists of a maximum of 65,536 words. Likewise, the maximum size of the data area for a given program is 65,536 words.

PROGRAM AREAS IN MEMORY



A code area is comprised of one or more procedures. A procedure is a block of machine instructions that can be called into execution to perform some specific task. A procedure (i.e., the block of instructions that a procedure represents) can be invoked (called) from any point in the program. When a procedure is called, the current environment is automatically saved by the hardware; when the procedure finishes, the previous environment is automatically restored. The procedure itself executes in an environment separate from other procedures.

The code part of a program is not modifiable. Therefore, all code is inherently sharable and reentrant.

T/TAL

Programs for the Tandem 16 are written in Tandem's Transaction Application Language (T/TAL). T/TAL is a high-level, block-structured, procedure-oriented language designed for ease of programming and efficient use of the many architectural features of the Tandem 16 (such as separate code and data).

A typical statement written in T/TAL looks like this:

```
IF item = taxable THEN payment := price + (price * taxrate);
```

The upper case elements are reserved words in T/TAL, the lower case elements are data variables, := means "is assigned the value of", and * means multiply.

Some characteristics of T/TAL are:

* Free-form Structure

  The free-form structure of T/TAL permits programmers to format their programs in a manner providing readability and self documentation.

* Machine Independent

  T/TAL programs are written using such high-level constructs as: IF THEN, WHILE DO, DO UNTIL, CASE, etc. The T/TAL compiler generates optimum code taking advantage of the Tandem 16's hardware characteristics.

  Features are provided in T/TAL for programmers desiring to explicitly operate on hardware registers (STACK and STORE statements) and program at an assembly language level (CODE statement).

* Identifiers

  Program elements such as constants, variables, labels, and procedures are identified throughout a source program by use of symbolic, programmer-assigned identifiers. This eliminates the need for a programmer to keep track of specific memory addresses. An identifier can contain up to 31 alphanumeric characters.

* Data types

  INT (integer), INT(32) (double word integer), STRING (byte), and FIXED (18-digit fixed point).

* Block (multiple element) Operations

  T/TAL provides multiple element operations such as move block, compare blocks, and scan block.

* Bit Operations

  T/TAL provides bit operations such as bit deposit, bit extraction, and bit shifts.

* Procedures

  As previously mentioned, a procedure is a block of machine instructions that exists only once in a program but can be called into execution from any point in the program. Procedures, as implemented in the T/TAL language, have special properties that make them nearly as versatile as complete programs. A program has

a global data area that is accessible only by statements within that program; a procedure has its own (local) data area that is accessible only by statements within that procedure. Unlike the program's global data area, however, a procedure's local data area is allocated and initialized only when the procedure executes. This provides two major benefits: 1) storage is not allocated except when needed (keeping the total amount of storage required by a program to a dynamic minimum) and 2) the data area is initialized when the procedure is entered.

The procedure oriented structure of T/TAL permits the programmer to separate a complex application program into relatively simple procedures. When designing a program, the programmer determines the basic operations to be performed, then writes procedures to perform each operation. Procedures are generalized in that the programmer need not be aware of the actual data variables involved when writing a procedure. Instead, the names of the actual variables can be passed to a procedure as parameters when the procedure is invoked.

* Recursive Procedures

  Because a procedure has its own local data area and the data area is initialized each time the procedure is entered, a procedure can call itself.

* Subprocedures

  Subprocedures are similar to procedures in that they can have their own variables and can be called recursively. However, a subprocedure is a part of a procedure and therefore can be called only from the procedure in which it resides.

* Structures

  T/TAL provides a feature - the STRUCT declaration - for describing and accessing a set of related data variables such as the fields of a file record STRUCT declarations are generated by the Data Definition Language (DDL) used for data base applications. This allows you to copy DDL-generated record descriptions into a T/TAL program.

  The STRUCT statement also provides a means for describing and accessing multi-dimensional arrays.

GUARDIAN

Programs execute in a processor module under control of Tandem's Guardian Operating System. Some functions that Guardian performs are: loading a program into the system for execution, bringing absent memory pages in from disc, and allocating processor module time among multiple processes. (The term "process" denotes a program running in a processor module.) This means that the programmer can write a program as though it will be the only program executing in a processor module and as though the entire program will always be resident in memory.

An additional function of Guardian is to continually check the integrity of the system. Periodically, Guardian in each processor module transmits an "I'm alive" message to all other processor modules in the system. Each processor module, in turn, periodically checks for receipt of an "I'm alive" message from every other processor module. If Guardian in a processor module finds that a message has not been received, it first verifies that it can transmit a message to its own processor module. If it can, it assumes that the non-transmitting processor module is malfunctioning. If it can't, it takes action to ensure that its own module does not impair the operation of other processor modules. In either case, Guardian then informs system processes and interested application processes of the failure.

File Management

An important service provided by Guardian is file management. File management is the means by which application programs perform input/output operations in the Tandem 16 computer system. A "file" can be all or a portion of a disc pack, a non-disc device such as a terminal or line printer, a process (i.e., running program), or the operator console. Files are identified by symbolic "file names". This frees the programmer from needing to know the physical addresses of i/o devices and permits addition and reconfiguration of input/output devices without the need to rewrite or recompile programs.

File management operations are performed by calling procedures that are part of the operating system. All files are accessed through this same set of procedures thereby providing a single, uniform access method. Additionally, the file management procedures are designed so as to mask the peculiarities of various devices.

Process Control

Process control is another important service provided by Guardian. Through the process control functions, an application process can run and stop processes in any processor module in the system and can monitor the operation of any processor module or any process running

in the system. If a module fails or a process stops, or if a failed module becomes operable, Guardian will notify the application process. Process control functions are invoked by making calls to operating system procedures.

Utility Procedures

Utility procedures are provided that convert numbers from internal machine representation (i.e., binary) to a human readable form, and provide the current time of day.

Checkpointing Facility

The Checkpointing Facility provides the capability for writing application programs that can recover from a failure of a processor module. To use the Checkpointing Facility, an application program is executed as a process-pair. (A process-pair is the execution of the same program in two separate processor modules. One member of a pair is designated the "primary"; it performs the work. The other member is designated the "backup"; it monitors the operation of the primary).

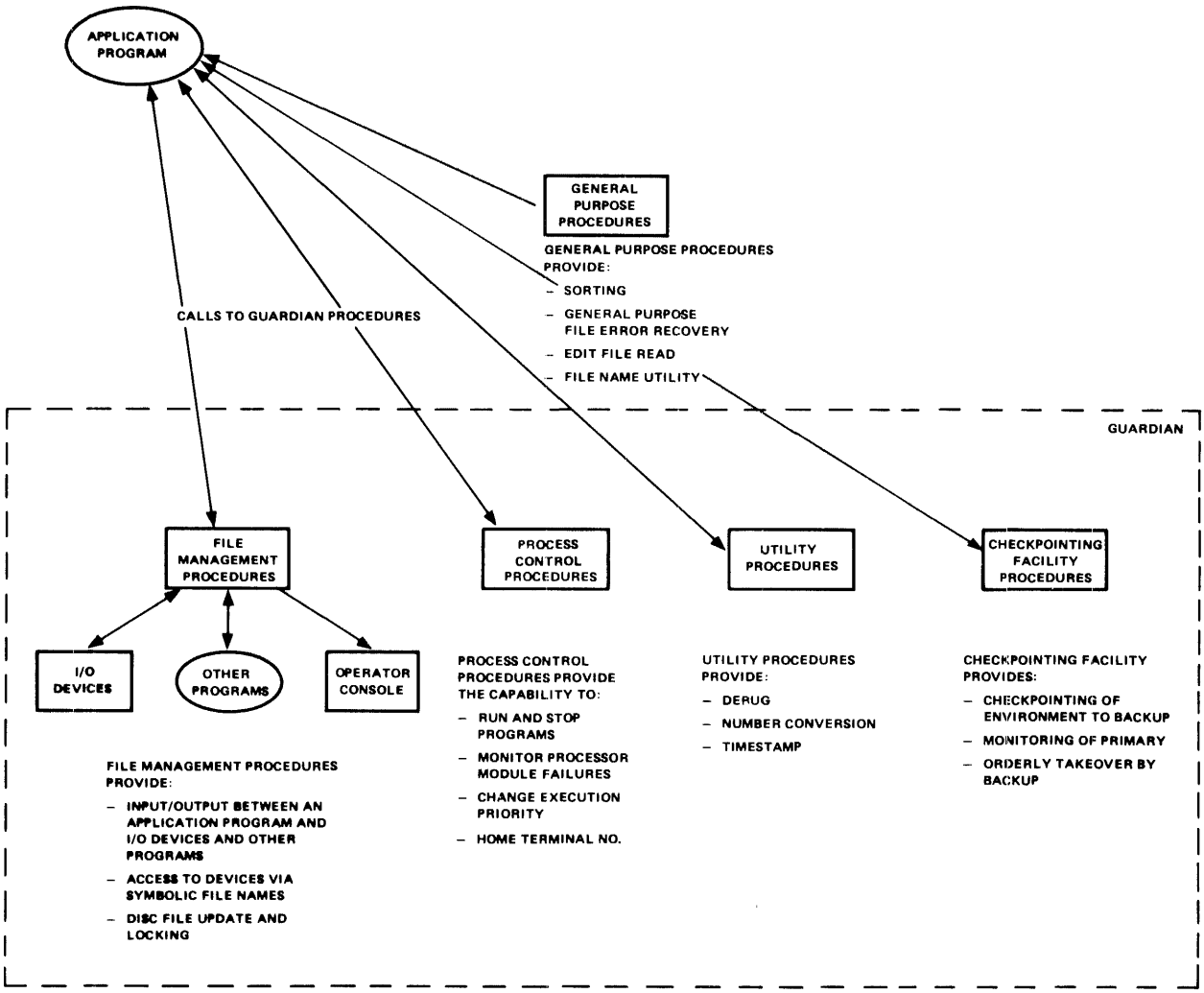
The Checkpointing Facility is used by a primary member of a pair to periodically send "checkpoint" information to its backup. The checkpoint information includes a "picture" of (all or a portion of) the primary's data area and may include control information regarding files in use by the primary. For the backup, the Checkpointing Facility is used to receive and process the checkpoint information and to monitor the operability of the primary's processor module. In the event that the primary's processor module fails, the Checkpointing Facility turns control over to the backup as indicated by the latest checkpoint.

GENERAL PURPOSE PROCEDURES

The general purpose procedures provide such functions as: expanding file names from the "external" form used by Tandem-supplied programs to the "internal" form used by the file system, a general purpose file management error recovery routine, a routine for reading from the specially formatted file used by the Tandem-supplied EDIT program.

The general purpose procedures are not strictly a part of Guardian. Rather, they are supplied in object form and, if one is used, it is either made part of the operating system's library of procedures (at system generation time) or made part of the application object program (via the UPDATE Program - see "Program Development Tools").

OPERATING SYSTEM SERVICES



PROGRAM DEVELOPMENT TOOLS

Five Tandem-supplied programs are provided that aid in program development:

* COMINT - the Command Interpreter,
* EDIT - the text editor;
* TAL - the T/TAL language compiler,
* XREF - the T/TAL cross reference generator, and
* UPDATE - the object file editor.

Additionally, DEBUG, the interactive debugging facility is provided.

Command Interpreter Program

The Command Interpreter (COMINT) is a program that is used interactively to run programs, check system status, create and delete disc files, and alter system hardware states. An important feature of the Command Interpreter is its ability to pass user specified parameter information to a program at run time (see Running the Object Program, below).

Text Editor Program

The text editor program (EDIT) is used to initially prepare source programs (written in T/TAL). The text editor is an interactive program that allows the programmer to enter and make changes to the source program through an interactive terminal. Text entered through the text editor is stored in a file on disc under a name given by the programmer. This name also specifies the source program to the T/TAL compiler and is used later if the programmer wishes to "edit" the file.

T/TAL Compiler Program

The T/TAL compiler program (TAL) reads source statements from one or more files and compiles the statements into a ready-to-run object program. Like the source program file, the object program file is given a name. The object program name is used to run or modify the program. As a by-product of the compilation, a completely annotated listing of the source program is provided. Certain listing options provide the machine instruction code generated and a map of all the identifiers used in the program.

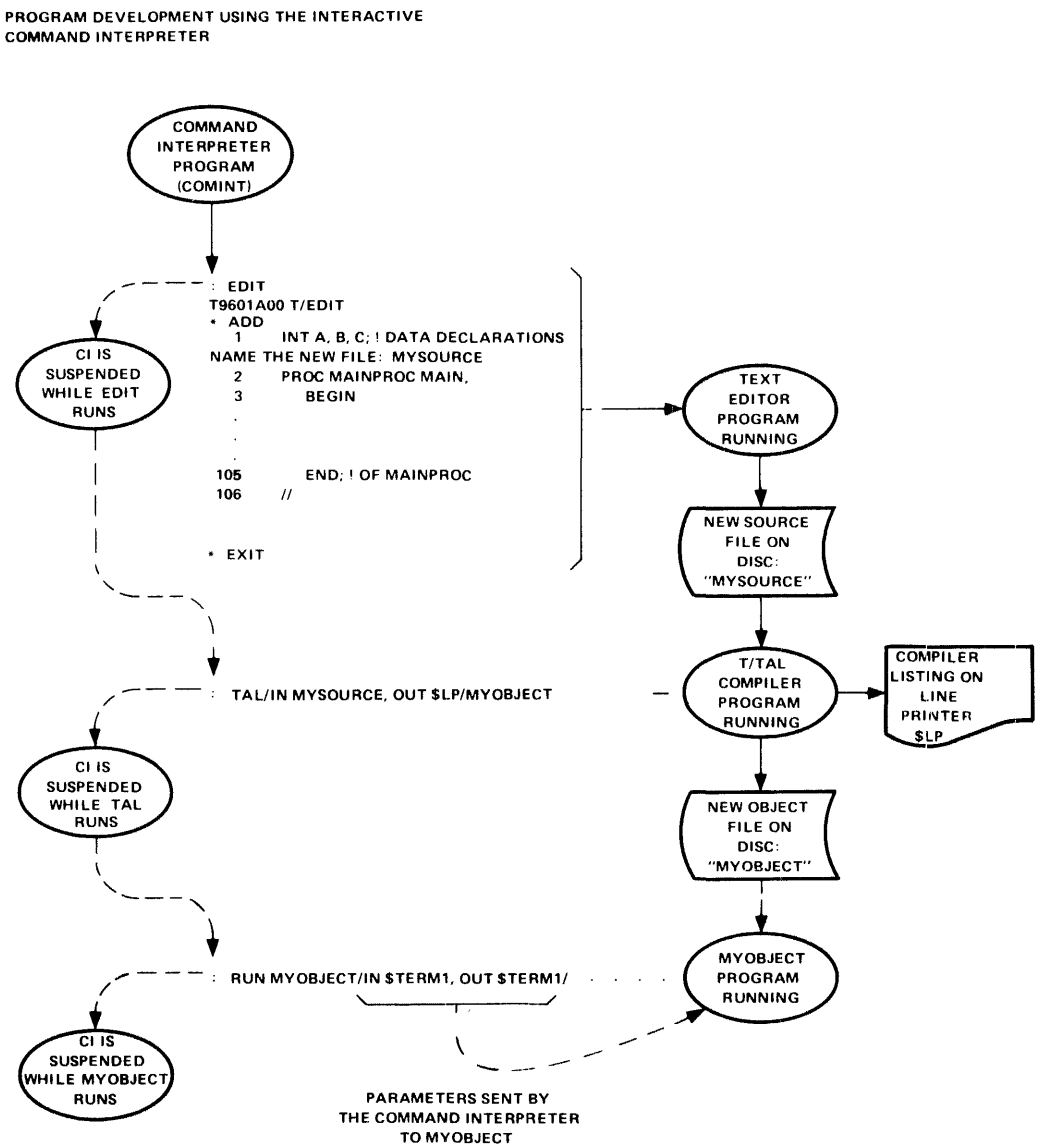
Cross Reference Program

The cross reference program (XREF) reads T/TAL source programs and provides a listing showing where each identifier in a program is used.

Object File Editor Program

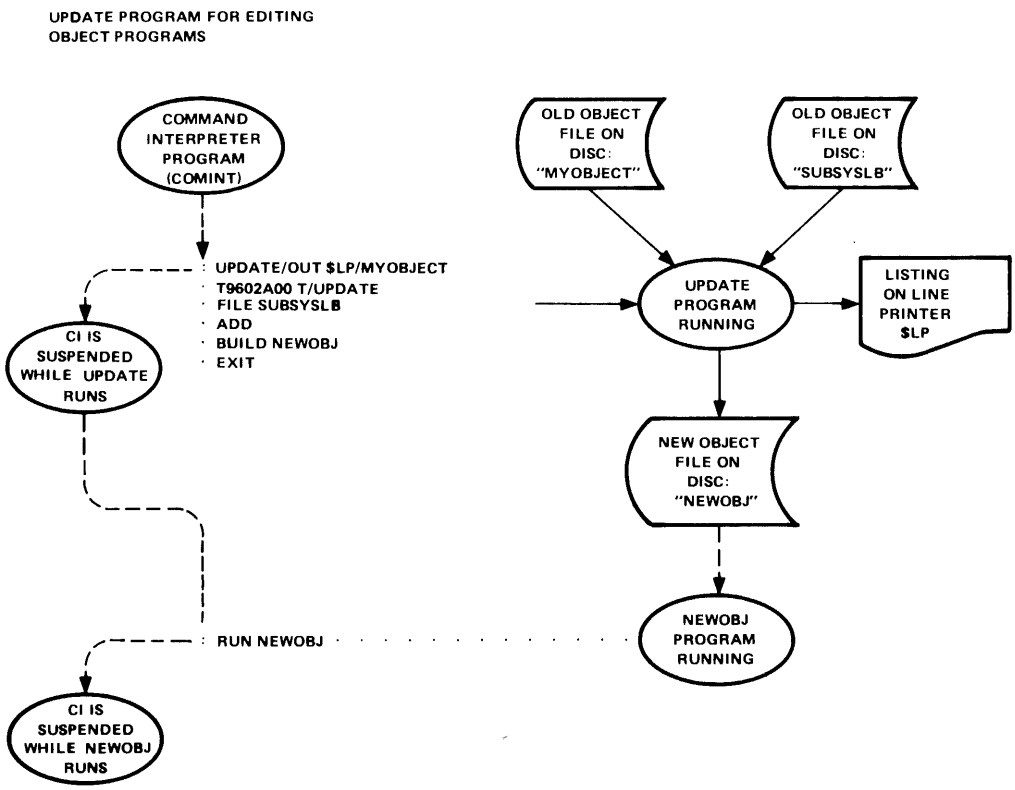
The object file editor (UPDATE) provides the capability to create new object program files from procedures existing in previously compiled object program files. Additionally, UPDATE can be used to display and modify the contents of existing object program files.

PROGRAM DEVELOPMENT



-->

PROGRAM DEVELOPMENT (cont'd)



Debugging Facility

The debugging facility (DEBUG), which is an integral part of the operating system, enables a programmer to isolate problems in an application program. Programs are debugged through an interactive terminal while running: breakpoints can be set and the values contained in variables can be displayed or modified.

RUNNING THE OBJECT PROGRAM

Programs are run in a Tandem 16 processor module when either the Command Interpreter RUN command is given, the process control NEWPROCESS procedure is called, or, if so specified, when a processor module is loaded and initialized.

To run a program using the Command Interpreter, the file name of the object program is given, and optionally, the priority to be assigned to the new process, the number of data pages required, the processor module where the program is to run, the input and output files, and a program-dependent parameter string to be passed to the new process.

For example, to run an application program called "myprog", the following run command might be given:

```
RUN myprog
```

During application process execution, certain abnormal conditions may occur. If this happens, a "trap" occurs. A trap causes the application process to enter either the debug state or, if one is specified, a trap handling mechanism in the application process. A trap is caused by any of the following conditions:

* An illegal memory address reference
* An illegal instruction specification
* An arithmetic overflow
* A memory stack overflow
* A memory manager (i.e., virtual memory) read error
* A non-availability of memory for overlay
* An uncorrectable memory error
* A map parity error

CONVENTIONS IN THIS MANUAL

Reserved Symbols and Syntactic Elements

Reserved symbols in T/TAL and calls to operating system procedures are shown in upper case. Syntactic entities are shown in lower case, surrounded by less-than and greater-than symbols <...>:

```
IF <conditional expression> THEN <statement> ELSE <statement>
```

IF, THEN, and ELSE are reserved symbols; <conditional expression> and <statement> are syntactic entities.

If a less-than or a greater-than symbol is required in the syntax, it is surrounded by quotations marks - "<"..."<":

```
<a>. "<" <b> : <c> ">"
```

here the "<" and ">" are required in the syntax. If <a> is an alphabetic character and <b> and <c> are digits, the following could be written

```
Z.<10:15>
```

Identifiers used in examples are shown in lower case, and when referenced in text are also surrounded by quotation marks "...".

Required Elements

Underlined elements are required:

```
<d> <e> [ <f>  ] ;
--- ---       -
 ^   ^        ^
 |   |        |
```

these elements are required.

Optional Elements

Elements that are surrounded by brackets [....] are optional:

```
<d> <e> [ <f> ] ;
          ^
          |
```

this element is optional.

If a left or right bracket is required in the syntax, it is surrounded by quotations marks - "["..."]":

```
<g> "[" <h> "]"
```

here the "[" and "]" are required in the syntax. If <g> is an alphabetic character and <h> is a digit, the following could be written

```
Z[10]
```

Choice of Elements

Required elements where a choice is involved are surrounded by braces:

```
{ <h> }
{ <i> }
{ <j> }
{ <k> }
```

means choose one from the list.

Optional elements where a choice is involved are surrounded by brackets:

```
[ <l> ]
[ <m> ]
[ <n> ]
[ <o> ]
```

means choose none or one from the list.

Element Lists

Where a list of like elements can be written, the syntactic element is followed by a "separator" symbol and an ellipsis consisting of three periods:

```
<p>  ...      a "blank" is the separator

<q> , ...     a "comma" is the separator

<r> ; ...     a "semicolon" is the separator
```

Each element of a list is separated from the other by the designated separator symbol. If the list is composed of none or one elements, the separator symbol is not used. The separator symbol does not follow the last element in the list.

For example, for

```
<q> , ...
```

where <q> is a digit, the following list could be written

```
1,2,3,4,5,6,7,8,9
```

If a list is formed of repeated groups of elements, the group that is repeated is surrounded by braces "{...}"; following the right brace is the separator symbol followed by three periods.

For example, for

```
{ <q> <s> } ; ...
```

where <q> is a digit and <s> is an alphabetic character, the following list could be written

```
1 A; 2 B; 3 C; 4 D; 5 E; 6 F; 7 G; 8 H
```

If the entire list is optional, it is shown in the form

```
[ <t> ... ]
```

or

```
[ { <u> <v> } ... ]
```

with the applicable separator symbol.

T/TAL

A T/TAL source program contains three basic elements:

"declarations",

"expressions", and

"statements".

DECLARATIONS

A declaration defines the use of an identifier. A declaration consists of:

* An identifier.
* A "class"
* An initialization value (optional in many cases).
* A terminating semicolon ";".

An "identifier" is a symbolic name, assigned by a programmer, that identifies an element used in a program. All identifiers must be defined before they can be used elsewhere in a program. An identifier consists of a maximum of 31 alphabetical and/or numerical characters. Circumflex symbols "^" can be included as part of an identifier.

i

is an identifier.

compute^number

is also an identifier. Here the circumflex symbol makes a single identifier appears as two words.

The "class" of an identifier defines the meaning that the identifier has when it is used elsewhere in the program. The classes are:

class	meaning
data variable	= one or more contiguous memory locations
LITERAL	= a constant numerical value
DEFINE	= a block of source text
LABEL	= a program location
PROC	= a procedure
SUBPROC	= a subprocedure
ENTRY	= a secondary entry point into a procedure or subprocedure
STRUCT	= a data variable grouping (see section 2.23)

A "data variable" is a word or group of words in memory from which values can be fetched and results of computations stored. Associated with a data variable is a data <type>. The <type> of a variable determines the hardware instructions (i.e., single word, double word, byte, etc.) that the compiler will emit when a variable is referenced in the source program. The <types> associated with data variables are:

<type>	description
INT	16-bit word
INT(32)	32-bit doubleword
STRING	8-bit byte
FIXED [(<fpoint>)]	64-bit quadrupleword

The class data variable is further defined as:

"simple" - contains one element of a specified data <type>. A simple variable is used to store a one element item such as the result of a calculation

"array" - contains multiple elements of a specified data <type>. An array variable is used to store a multiple element item such as a string of characters

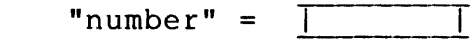
"pointer" - contains the address of another data variable. Referencing a pointer accesses the variable whose address is contained in the pointer. The variable is treated as the data <type> declared for the pointer. A pointer is typically used when it is desirable to use a single identifier to access many different data variables during the execution of a program

Some examples of declarations:

To define an identifier as a simple (one element) variable having a data type of INT (16-bit word), the following declaration could be written in a source program:

```
INT number;
```

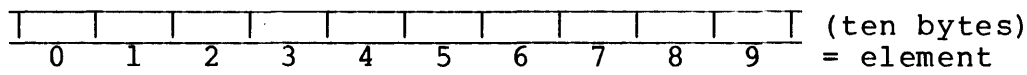
defines "number" to be a <type> INT variable and allocates one word of memory for the variable "number":



To define an identifier as an array of variables having a data type of STRING (8-bit byte) consisting of ten elements, the following declaration could be written:

```
STRING bytes[0:9];
```

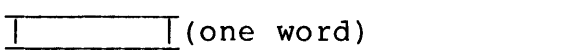
"[0:9]" defines the length and the first and last elements of the array "bytes" and allocates five words of memory for the variable "bytes".



To define an identifier as a pointer variable having a data type of STRING, the following declaration could be written:

```
STRING .spointer;
```

"." indicates that the contents of the variable "spointer" are to be used as the indirect address of another, as yet unspecified, variable. (A pointer variable must be initialized with the address of another variable before its use has any meaning.) One word is allocated for a pointer variable.

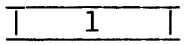


Data variables can be given initial values when declared. This is accomplished by following the name of the variable with the symbol ":=" (the assignment operator) followed by the initializing value.

Some examples:

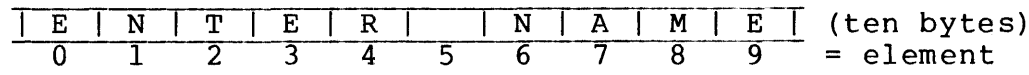
```
INT count := 1;
```

assigns the initial value of 1 to the variable "count".




```
STRING buffer[0:9] := "ENTER NAME";
```

assigns the initial value "ENTER NAME" to the array variable "buffer".




```
STRING .sp := @bytes;
```

assigns the address of the array "bytes" to the pointer "sp". Preceding the name of a variable with a commercial at symbol "@" obtains the address of that variable.

The LITERAL declaration is used to associate a 16-bit value with an identifier:

```
LITERAL minute = 60;
```

The identifier "minute" has the value 60 when used in the program.

A procedure declaration assigns an identifier to a procedure:

```
PROC compute^value;
  BEGIN
    .
    .
  END;
```

Using the identifier "compute^value" elsewhere in the program causes the procedure to be executed.

EXPRESSIONS

There are two kinds of <expressions> used in T/TAL

<arithmetic expressions> and <conditional expressions>

An <arithmetic expression> is a rule (i.e., formula) for computing a numeric value. An <arithmetic expression> consists of one or more operands and arithmetic operators:

<operand> <arithmetic operator> <operand> ...

An operand can be a variable, a part of a variable, a constant, or a function. (A function is a program element that produces a value of a specific data <type> when referenced in an expression. The return value is usually the result of an operation involving one or more variables.) An arithmetic operator specifies the kind of arithmetic operation to be performed. The basic <arithmetic operators> are

+	-	addition
-	-	subtraction
*	-	multiplication
/	-	division

Some examples of arithmetic expressions:

expression	description
number + 2	the contents of the variable "number" plus 2
4 * 3 - 2	four times three minus two.
20/4	twenty divided by four.

The language permits arithmetic operations on any data type, but does not permit operands having different data types to be mixed in the

same expression. There are, however, a number of standard functions for treating an operand of one data type as another data type.

A <conditional expression> is a rule for establishing the relationship between two operands. The result of a conditional expression is either a true or a false state. The result is used to control the flow of program execution.

A <conditional expression> is made up of one or more "conditions". A condition is:

<left operand> <relational operator> <right operand>

An operand may be a variable (including arrays), a constant, or a function. The relational operator defines the relationship between the two operands that will produce a true state. The basic <relational operators> are:

=	-	the values of the operands are equal
<	-	the value of the left operand is less than that of the right operand
>	-	the value of the left operand is greater than that of the right operand
<=	-	the value of the left operand is less than or equal to that of the right operand
>=	-	the value of the left operand is greater than or equal to that of the right operand
<>	-	the values of the operands are not equal

A condition is tested for a false state rather than a true state if it is preceded by the "NOT" operator. Several conditions can be combined into one expression using the "AND" and "OR" operators. The use of the AND operator means that the adjacent conditions must both be true for the expression to be true; the use of the OR operator means that either adjacent condition can be true.

There is also a form of "condition" that tests a single operand for a true (i.e., non-zero) state and another form that tests the state of certain hardware indicators.

Some examples of conditional expressions:

expression	description
a < b	the expression is true if "a" is less than "b"
a	the expression is true if "a" has a nonzero value.
a AND b	the expression is true if both "a" and "b" have nonzero values.
NOT a OR b	the expression is true if "a" has a value of zero or "b" has a nonzero value.

STATEMENTS

A T/TAL <statement> is an order to perform some action. A statement consists of:

* a reserved symbol (i.e., a word or group of special characters) that identifies the action to be performed.

* One or more identifiers indicating program elements that are to be used or that are affected when the statement is executed.

* In many cases, an <arithmetic expression>.

* In many cases, a <conditional expression>.

The T/TAL statements are:

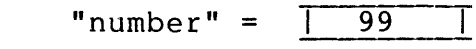
assignment	-	stores a value into a variable
compound	-	groups multiple statements together so that they will be executed as a block of statements
GOTO	-	directs the flow of program execution to a labeled statement
IF	-	causes a statement to be executed if a specified condition is true
CASE	-	selects one of a set of statements to be executed depending on the value of an expression
FOR	-	repeatedly executes the same statement a specified number of times
WHILE	-	repeatedly executes the same statement while a specified condition is true
DO	-	repeatedly executes the same statement until a specified condition becomes true
move	-	moves a block of elements from one location to another
SCAN	-	scans a block of elements for a specified character
CALL	-	invokes a procedure or subprocedure
RETURN	-	returns to the caller from a procedure or subprocedure

Some examples of T/TAL statements:

The assignment statement is used to store a value into a data variable:

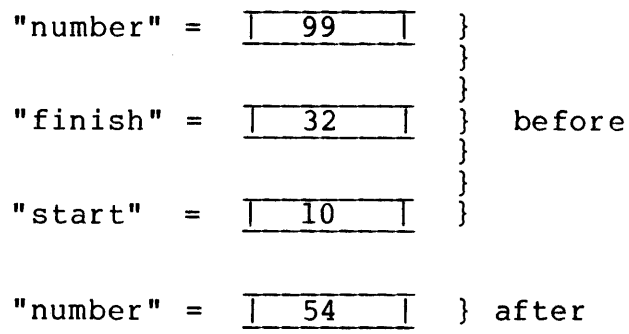
```
number := 99;
```

":=" is the symbol identifying the assignment statement. In this case, the value 99 is stored in the data variable "number".




```
number := finish * 2 - start;
```

"finish * 2 - start" is an <arithmetic expression>. The value represented by the identifier "finish" is multiplied by two; the value represented by the identifier "start" is then subtracted. The result is stored in the data variable "number".



The "compound statement" is used to group a number of T/TAL statements together to be treated as if they formed a single statement:

```
BEGIN
  <statement>;
  <statement>;
    .
END;
```

The BEGIN-END pair brackets the <statements> to form a single statement known as a compound statement.

A GOTO statement is used to direct the flow of program execution to another point in a program:

```
GOTO next;
  .
  .
  .
  .
  .
next: a := 1;
```

GOTO is the symbol identifying the GOTO statement. "next" is called a <label> and identifies a <statement> elsewhere in the program.

The IF statement uses a <conditional expression> to control the flow of program execution:

```
IF a = b THEN GOTO next;
   .
```

"a = b" is a <conditional expression>. "IF .. THEN" are symbols identifying an IF statement. In this case, IF the value represented by the identifier "a" is equal to the value of the identifier "b" the conditional expression is true and the GOTO

statement is executed. If "a" did not equal "b" the next statement in the program would be executed (the GOTO statement would be bypassed).

There are other statements that use <conditional expressions> to control the flow of program execution. For example, the WHILE statement forms a program loop:

```
WHILE a < 10 DO
  BEGIN
    n := n - a;
    a := a + 1;
  END;
```

The statements in the <compound statement> formed by the BEGIN-END pair are executed until the value of "a" equals 10.

The DO statement also forms a program loop:

```
DO
  BEGIN
    .
    .
  END
UNTIL b > a;
```

The compound statement is executed until the value of the identifier "b" is greater than the value represented by "a".

The FOR statement is used to repeatedly execute the same statement a specified number of times:

```
FOR index := 0 TO 12 BY 1 DO array[index] := " ";
```

The data variable "index" is referred to as the "step" variable and is assigned the initial value 0. The value "12" is referred to as the "limit" value.

At the beginning of the FOR loop, the step variable "index" is tested to see if it exceeds the "limit" value (12 in this case). If the step variable is less than or equal to the limit value, the statement following DO (i.e., the assignment statement) is executed. If the step variable "index" exceeds the limit value 12, FOR statement execution is completed and the next statement in the program is executed.

At the end of the FOR loop, after the assignment statement is executed, 1 is added to the contents of the step variable "index". Program execution returns to the beginning of the FOR loop where the new value of "index" is compared with limit value 12. If "index" is less than or equal to 12, the assignment statement is executed again (using the new value of "index"). When "index" exceeds 12, the next statement in the program is executed.

The move statement is used to move a block of data from one array to another:

```
outbuffer ':=' inbuffer FOR 72;
```

"':='" is one of the symbols identifying a move statement. In this example, 72 elements of an array "inbuffer" are moved into an array "outbuffer".

The scan statement is used to search an array for a particular character:

```
SCAN inbuffer UNTIL " " -> @nonblank;
```

scans the array "inbuffer" until an ASCII blank character is encountered. When a blank character is encountered, its address is put into the address pointer "nonblank" and the statement following the SCAN statement is executed.

The CALL statement is used to invoke a procedure or subprocedure:

```
CALL compute^value;
```

"compute^value" is an identifier assigned to a procedure. When "compute^value" completes, the statement following the CALL statement is executed.

PROGRAM ORGANIZATION

In its most basic form, a T/TAL program is comprised of

"global declarations" and "procedure declarations".

```
[ global declaration ]          ! optional.
          .
          .
[ global declaration ]

[ procedure declaration ]       ! optional.
          .
          .
[ procedure declaration ]

MAIN procedure declaration      ! required.
-------------------------

valid global declarations are

 - data declaration
 - LITERAL declaration
 - DEFINE declaration

procedure declaration is

  PROC <name> [ ( <formal parameters> ) ] [ MAIN ] ;
  ---- ------                                     -
     [ <parameter specifications> ; ]

    BEGIN
    -----
      [ local declaration ]
                .
                .
      [ local declaration ]

      [ subprocedure declaration ]
                .
                .
      [ subprocedure declaration ]

      [ [ <statement> ] ; ]
                .
                .
      [ [ <statement> ] ; ]

    END ;
    --- -
```

```
valid local declarations are

- data declaration
- LITERAL declaration
- DEFINE declaration
- ENTRY declaration

subprocedure declaration is

  SUBPROC <name> [ ( <formal parameters> ) ] ;
  ------- ------                              -
     [ <parameter specifications> ; ]

    BEGIN
    -----
      [ sublocal declaration ]
                  .
                  .
      [ sublocal declaration ]

      [ [ <statement> ] ; ]
                  .
                  .
      [ [ <statement> ] ; ]

    END ;
    --- -

      valid sublocal declarations are

      - data declaration
      - LITERAL declaration
      - DEFINE declaration
      - ENTRY declaration

<statement> is

- assignment
- compound
- GOTO
- IF
- CASE
- FOR
- WHILE
- DO
- move
- [R]SCAN
- CALL
- RETURN
```

Global Declarations

The global declarations assign identifiers to program elements that are to be referenced throughout the program. The program elements that can be declared globally are:

- data variables
- LITERALs
- DEFINEs

Procedure Declarations

The procedure declarations contain the program's executable parts. A program may contain many procedures but must contain at least one procedure. One procedure must be declared as being the "MAIN" procedure. (The MAIN procedure is the first one executed when the program is run.)

A procedure declaration consists of a

procedure heading	- assigns a name to a procedure - describes the procedure - defines its parameters

and a

procedure body.	- may contain local declarations - may contain subprocedures - contains statements (that are compiled into instruction codes)

A procedure heading is of the general form:

```
PROC <name> [ MAIN ] ;
---- ------          -
```

<name> is the identifier to be assigned to the procedure.
"MAIN" indicates that the procedure is the main procedure.

A procedure body is of the general form:

```
BEGIN
-----
  [ local declaration ]
            .
            .
  [ local declaration ]

  [ subprocedure declaration ]
            .
            .
  [ subprocedure declaration ]

  [ [ <statement> ] ; ]
            .
            .
  [ [ <statement> ] ; ]

END ;
--- -
```

LOCAL DECLARATIONS: The "local declarations" (optional) define program elements that can be referenced only within the procedure's body. The purpose of local declarations is to permit each procedure to have program elements (e.g., variables) that are completely separate from other procedures. Identifiers declared locally can be referenced only by statements within the procedure body (and by subprocedures within the same procedure body). Local identifiers cannot be referenced directly by other procedures. (Local identifiers can be passed as parameters to other procedures however.)

Program elements that can be declared locally are:

- data variables
- LITERALs
- DEFINEs
- ENTRY points

An example of a local declaration within a procedure declaration is:

```
PROC a;
  BEGIN
    INT b,c,d; ! local declaration.
      .
      .
      .
  END;
```

SUBPROCEDURE DECLARATIONS: The "subprocedure declarations" (optional) define the procedure's subprocedures. A subprocedure declaration assigns a name to a subprocedure, defines characteristics of the subprocedure and its parameters, and contains the statements that are compiled into instruction codes. Subprocedures are discussed later.

STATEMENTS: A procedure is called into execution through use of a CALL statement. Procedure execution begins with the first statement of the procedure body. Statements are executed until the "END;" of the procedure body is encountered. At this point, program execution returns to the statement following the call to the procedure:

```
CALL a; ---------------          PROC a;
<statement>; <-----   |            BEGIN
                  |   |              INT b,c,d;
                  |   |                .
                  |   ------------> <statement> ; ! first statement.
                  |                  <statement> ;
                  |                    .
                  |                    .
                  |                  <statement> ; ! last statement.
                  -------------   END;
```

Additional "return" points can be specified in a procedure body through use of the RETURN statement:

```
CALL a; ---------------          PROC a;
<statement>; <-----   |            BEGIN
                  |   |              INT b,c,d;
                  |   |                .
                  |   ------------> <statement> ; ! first statement.
                  |                  <statement> ;
                  |                    .
                  |                  IF b = c THEN
                  |<- - - - - - - - -  RETURN;
                  |                    .
                  |                  IF d = 0 THEN
                  |<- - - - - - - - -  RETURN;
                  |                    .
                  |                  <statement> ; ! last statement.
                  -------------- END;
```

The statements within a procedure can reference identifiers declared within that procedure as well as identifiers declared globally. This means that statements within a procedure can call subprocedures declared in that procedure's body and can call any procedure in the program. Statements within a given procedure body cannot reference identifiers declared in other procedures or in subprocedures.

SUBPROCEDURES: Subprocedures are similar to procedures in that they can be called into execution from statements within the procedure body and that they have their own (sub)local variables. Like procedures, subprocedures can be passed parameters and can return results. Subprocedures, however, have limited sublocal storage capabilities and cannot be called from other procedures.

A subprocedure is of the general form:

```
SUBPROC <name> ;
BEGIN
  [ sublocal declaration ]
              .
              .
  [ sublocal declaration ]

  [ [ <statement> ] ; ]
              .
              .
  [ [ <statement> ] ; ]
END ;
```

The "sublocal declarations" (optional) are similar to a procedure's local declarations; they define program elements that can be referenced only by statements within the subprocedure body. Program elements that can be declared sublocally are:

- data declaration
- LITERAL declaration
- DEFINE declaration
- ENTRY declaration

The statements with a subprocedure can reference identifiers declared within that subprocedure's body, identifiers declared locally in its procedure (including the procedure's parameters, if any), and identifiers declared globally. Therefore, statements within a subprocedure can call subprocedures declared within its procedure and can call any procedure in the program.

PARAMETERS: Procedures and subprocedures can be passed parameters. Parameters are defined where a procedure (or subprocedure) is declared. The general form of a procedure having parameters is:

```
PROC <name> ( <formal parameter names> ) ;
   <parameter specifications> ;
BEGIN ! procedure body.
  .
  .
END;
```

The <formal parameter names> assign identifiers to the procedure's parameters. The <parameter specifications> describe each parameter as to its data <type> and whether the parameter is passed by "value" or by "reference" (value and reference are described in section 2.15, "Procedure Declaration").

For example:

```
PROC a (x,y,z);  ! formal parameter names are between parentheses.
   INT x,y,z;    ! parameter specifications.

  BEGIN
    INT num;
      .
    num := x * y + z;
      .
  END;
```

"x", "y", and "z" are the formal parameter names.

Formal parameters can be thought of as actual variables when writing a procedure. A formal parameter has meaning only within the body of a procedure.

When a call to a procedure is written, the programmer substitutes the names of actual variables in place of the formal parameters.

```
INT f,g,h;  ! actual variables

   .
CALL a (f,g,h);
   .
```

Invokes the procedure "a". "f", "g", and "h" are the actual names of variables that are passed to the procedure. Calling procedure "a" with those actual parameters is equivalent to writing the statement

```
num := f * g + h;
```

except that the instruction codes for procedure "a" exist only once in the object program.

The same procedure can be called for other sets of variables:

```
   .
CALL a (3,4,5);
   .
```

In this case, constants are substituted for the formal parameters. This call to procedure "a" is equivalent to writing the statement

```
num := 3 * 4 + 5;
```

FUNCTIONS: A procedure or subprocedure can be declared to be a function. A function returns a value of a specific data <type> when invoked (a function is invoked when it is referenced in an arithmetic expression). The general form of a function procedure is:

```
<type> PROC <name> ;
------ ---- ------ -
  BEGIN ! procedure body.
  -----
      .
      .
    RETURN <expression> ; ! the return value.
    ------ ------------ -
  END ;
  --- -
```

The <type> indicates the data <type> of the value that the function returns when it is invoked. Like a non-function (sub)procedure, execution begins with the first statement of the (sub)procedure body. Unlike a non-function (sub)procedure, a function (sub)procedure must have at least one RETURN statement (it can have more). The RETURN statement is used to return the value that the function produces (as well as return to the point where the function was invoked); <expression> is the value to be returned. Functions can also have parameters.

An example of a function procedure:

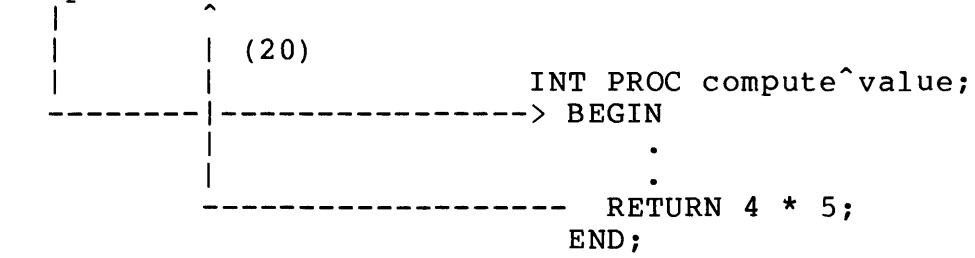
```
INT PROC compute^value;
  BEGIN
      .
      .
    RETURN 4 * 5;
  END;
```

The procedure name is "compute^value". It returns a <type> INT value when invoked. The return value is a result of the expression "4 * 5".

The function "compute^value" is then invoked by using its name in an arithmetic expression:

```
number := compute^value + 2;
```

"compute^value" executes when the arithmetic expression "compute^value + 2" is evaluated.



The result of the arithmetic expression "compute^value + 2", 22, is stored in the variable "number".

COMMENTS

Textual comments can be freely embedded in source language programs. A comment begins with an exclamation point !

```
! this is a comment.
```

On a line, all text to the right of an explanation point is treated as a comment unless the comment is terminated by another exclamation point

```
IF a = b THEN ! they are equal ! GOTO finish
```

causes the compiler to treat the text "! they are equal !" as a comment; other text on the line is compiled.

Comments are not continued from line to line; each comment in a new line must begin with an exclamation point.

EXAMPLE PROGRAM

The following example program is used to show how the basic constructs of TAL are used to form a complete program. It consists of one procedure called "main^proc". "main^proc" is designated a main procedure, therefore program execution begins with its first executable statement.

The example program executes as follows:

* The home terminal associated with the program's execution is opened
* The user is prompted to enter a string into the home terminal
* The string is read
* The program scans the string for an asterisk "*". If one is found, its location in the string is calculated, and a circumflex symbol is printed directly underneath.
* The program repeats indefinitely.
* The program is stopped by typing the "break" key on the terminal, then entering a Command Interpreter "STOP" command.

```
! EXAMPLE PROGRAM

INT  hometerm,      ! file number of home terminal
     left^side,     ! sbuffer address of asterisk
     num^xferred,   ! number of bytes transferred by file system
     count,         ! general purpose variable
     asterisk,      ! location of asterisk
     buffer[0:40]; ! input/output buffer

STRING
    .sbuffer := @buffer '<<' 1,  ! string pointer to i/o buffer
     blanks[0:71] := 72 * [" "]; ! blanks for initialization

?SOURCE $SYSTEM.SYSTEM.EXTDECS(MYTERM,OPEN,WRITEREAD,WRITE,STOP)
   ! Guardian procedure declarations

PROC main^proc MAIN;

  BEGIN

    CALL MYTERM(buffer); ! get name of home terminal.
    CALL OPEN(buffer, hometerm); ! open the home terminal.

    WHILE 1 DO ! infinite loop
      BEGIN
        sbuffer ':=' "ENTER STRING" -> left^side;
        CALL WRITEREAD(hometerm, buffer, 12, 72, num^xferred);
        sbuffer[num^xferred] := 0; ! delimit the input
        SCAN sbuffer UNTIL "*" -> asterisk; ! scan for asterisk
        IF NOT $CARRY THEN  ! asterisk found
          BEGIN
            sbuffer ':=' blanks FOR
               (count := asterisk - @sbuffer +
                         (left^side - @sbuffer));
            sbuffer[count] := "^";
            CALL WRITE(hometerm, buffer, count + 1);
          END;
      END;
  END;
```

The example program can be broken down to illustrate the basic constructs used in T/TAL.

In the example program, the global data declarations are:

```
INT  hometerm,
     left^side,
     num^xferred,
     count,
     asterisk,
     buffer[0:40];
```

```
   STRING
       .sbuffer := @buffer '<<' 1,
        blanks[0:71] := 72 * [" "];
```

There is one procedure declaration. It is

```
   PROC main^proc MAIN;
     BEGIN
        .
        .
     END;
```

There are no local data declarations.

The statements are:

```
       CALL MYTERM(buffer);
       CALL OPEN(buffer, hometerm, 0);
       WHILE 1 DO
         BEGIN
           sbuffer ':=' "ENTER STRING" -> left^side;
           CALL WRITEREAD(hometerm, buffer, 12, 72, num^xferred);
           sbuffer[num^xferred] := 0;
           SCAN sbuffer UNTIL "*" -> asterisk;
           IF NOT $CARRY THEN
             BEGIN ! compound statement.
               sbuffer ':=' blanks FOR
                  (count := asterisk - @sbuffer +
                            (left^side - @sbuffer));
               sbuffer[count] := "^";
               CALL WRITE(hometerm, buffer, count + 1);
             END;
         END;
```

A compound statement is formed by

```
       WHILE 1 DO
         BEGIN
           sbuffer ...
             .
             .
         END;
```

Within the above compound statement, another compound statement is formed by

```
           IF NOT $CARRY THEN
             BEGIN
               sbuffer ':=' ...
                 .
                 .
             END;
```

An arithmetic expression used in an assignment statement is

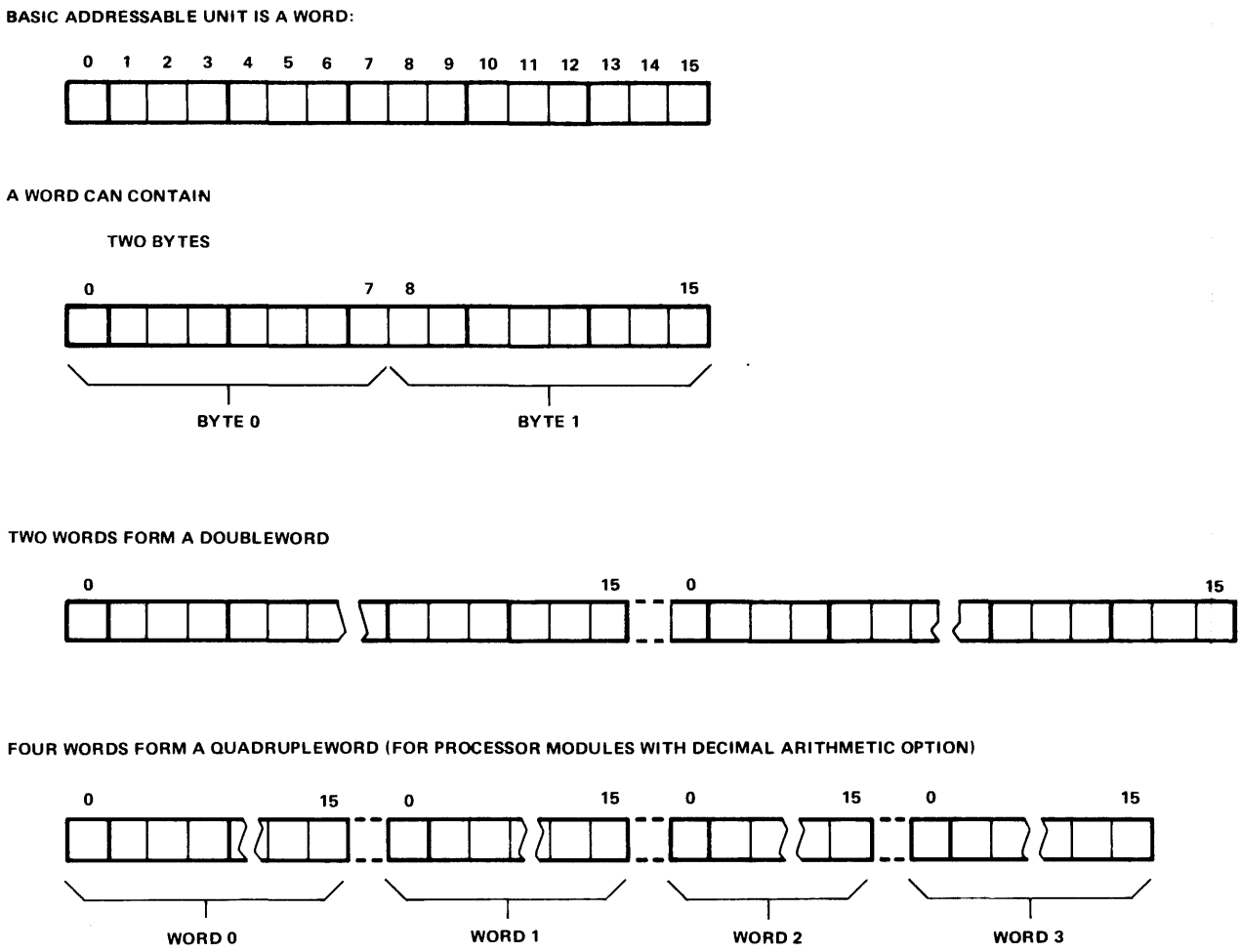
```
            (count := asterisk - @sbuffer + (left^side - @sbuffer));
```

The calls to Guardian procedures are

```
      CALL MYTERM(buffer);
      CALL OPEN(buffer, hometerm);
          .
          .
          .
          CALL WRITEREAD(hometerm, buffer, 12, 72, num^xferred);
          .
          .
          .
          CALL WRITE(hometerm, buffer, count + 1);
```

The basic unit of information in the Tandem 16 is the 16-bit word. In T/TAL, individual access to and operations on single or multiple bits (bit fields) in a word, 8-bit bytes, 16-bit words, 32-bit doublewords, and 64-bit quadruplewords are provided.

DATA FORMATS



BITS

The individual bits in a word are numbered from zero (0) through fifteen (15), from left to right:

```
                          1 1 1 1 1 1
word: 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
```

The following notation is used in this manual (and in the T/TAL language) to describe bit fields:

```
<identifier>.<left bit:right bit>
```

For example, to indicate a field starting with bit four and extending through bit 15 of a variable called "var", the following notation would be used:

```
var.<4:15>
```

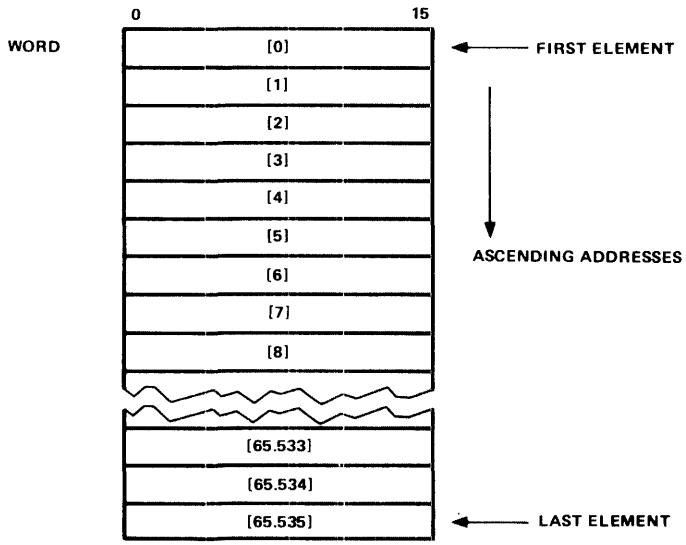
Or to indicate just bit 0 (zero), the following is used:

```
var.<0>
```

WORDS

The 16-bit word defines the Tandem 16's machine instruction length and its logical addressing range. The 16-bit word is the basic addressable unit stored in memory. The first word in logical memory is addressed as word[0], the last addressable location is word[65,535].

WORD ADDRESSES



In this manual, a number surrounded by brackets is used to denote an individual element (i.e., word, doubleword, byte, or quadrupleword) in a block (i.e., array) of elements:

```
<identifier>[element]
```

For example, to indicate the fourth element in an array called "array", the following notation is used:

```
array[4]
```

For purposes of illustration, the following notation is used in this manual (but not in the T/TAL language) to denote a block of elements:

```
<identifier>[first element:last element]
```

For example, to indicate the second through twentieth words in a block, the following notation is used:

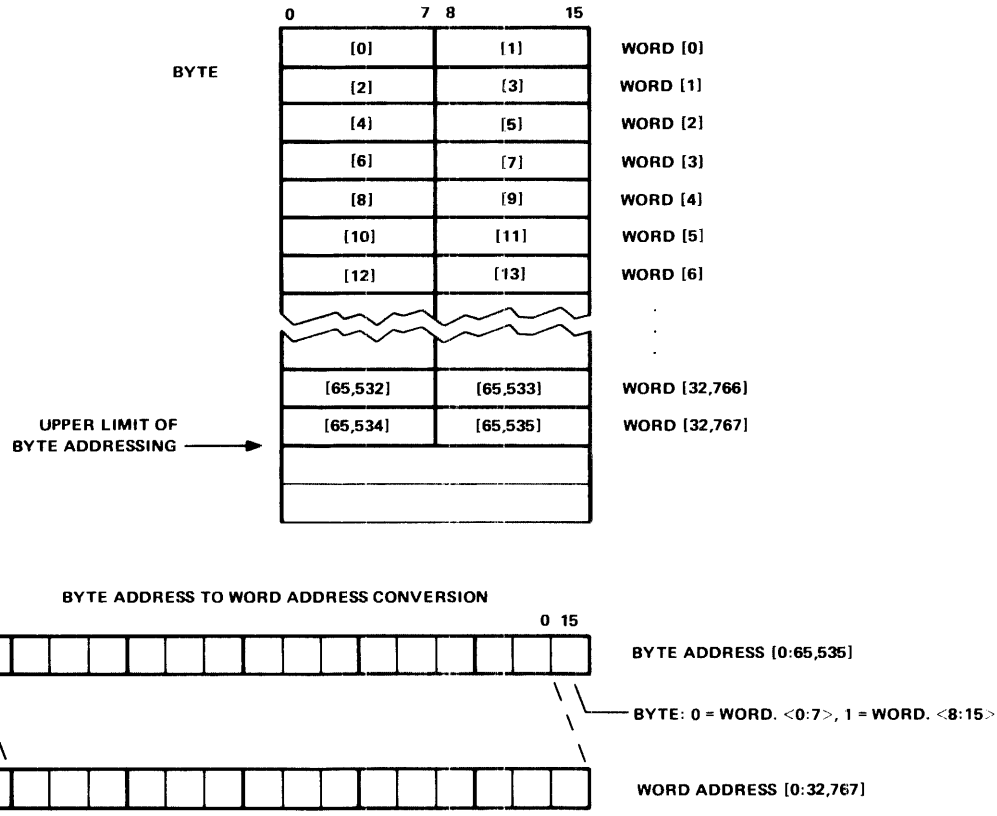
```
array[2:20]
```

BYTES

The 16-bit word has the capability to store two bytes. The most significant byte in a word occupies word.<0:7> (left half); the least significant byte occupies word.<8:15>.

The 16-bit address provides for element addressing of 65,536 bytes. Byte locations are addressed starting at byte[0] and extend through byte[65,535]. Two bytes are stored per word, therefore only the first 32,768 words of logical memory can be addressed on byte boundaries. The CPU converts a byte address to a word and element address as shown in the following illustration:

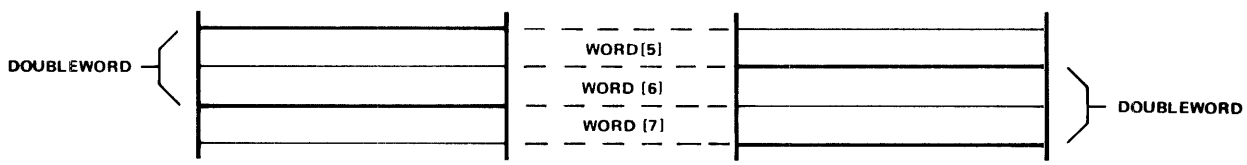
BYTE ADDRESSES



DOUBLEWORDS

Two 16-bit words can be referenced as a single 32-bit element. Doubleword elements are addressed on word boundaries, therefore doubleword addressing is permitted in all 65,536 words of logical memory.

DOUBLEWORD

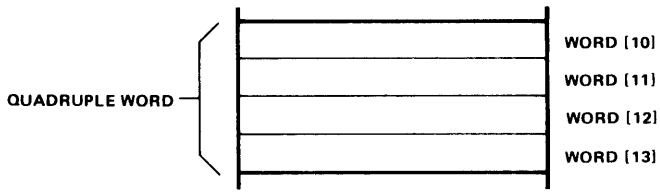


QUADRUPLEWORDS

Four 16-bit words can be referenced as a single 64-bit element. Quadrupleword elements are addressed on word boundaries, therefore quadrupleword addressing is permitted in all 65,536 words of logical memory.

QUADRUPLEWORD

A QUADRUPLEWORD CONSISTS OF ANY FOUR CONSECUTIVE MEMORY LOCATIONS



The Tandem 16 hardware, when performing arithmetic, can treat an operand as either a signed or unsigned number. Signed numbers are represented in 16 bits (a word), 32 bits (doubleword), or 64 bits (quadrupleword). Representation of unsigned numbers is restricted to 8- and 16-bit quantities.

Positive numbers are represented in true binary notation. Negative numbers are represented in two's complement notation with the sign bit (word.<0>) of the most significant word set to one. The two's complement of a number is obtained by inverting each bit position in the number then adding a one in word.<15> (with carry propagated through the word). For example, in 16 bits, the number 2 is represented:

```
                              1 1 1 1 1 1
  word: 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5

        0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
```

And the number -2 is represented

```
        1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0
```

The range of numbers that a variable can represent is determined by the data <type> assigned when the variable is declared. Four types of data variables are defined in T/TAL: single word integer, double word integer, byte, and four word fixed point.

SINGLE WORD INTEGER

Single word integer variables are defined as <type> INT. An INT variable can represent signed integers in the range of

-32,768 to +32,767

or unsigned integers in the range of

0 to +65,535.

Whether an INT is treated as a signed or unsigned value is determined by the <arithmetic operator> used when a calculation is performed. Signed arithmetic is indicated by using the basic operators

+ - * / (add, subtract, multiply, divide)

Unsigned arithmetic is indicated by surrounding an arithmetic operator with apostrophes

'+' '-' '*' '/'

Unsigned arithmetic should be used when performing calculations involving addresses.

INT variables can also represent the logical FALSE (zero) or TRUE (nonzero) states.

DOUBLE WORD INTEGER

Double word integer variables are defined as <type> INT(32). An INT(32) variable can represent signed integers in the range of

-2,147,483,648 to +2,147,483,647

or 9+ digits.

BYTE

Byte variables are defined as <type> STRING. A STRING variable represents unsigned values that can be contained in eight bits. The range of representable unsigned numbers is 0 to 255 (decimal). This of course includes the ASCII character set. STRING variables can also represent the logical FALSE and TRUE states.

FOUR-WORD FIXED POINT

Four word fixed point variables are defined as <type> FIXED. A FIXED variable can represent any 19-digit number in the range of

-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807.

All or part of a FIXED variable, at the programmer's discretion, can be treated as a fraction. The implied position of the decimal point is indicated by the (optional) <fpoint> specifier when a FIXED variable is declared (the word "implied" is used because a FIXED variable is represented internally as a 64-bit integer):

FIXED (<fpoint>)

where <fpoint> is a number from -19 to +19.

If <fpoint> is a positive value, it indicates the number of implied fractional digit(s) to the right of the decimal point. For example, an <fpoint> of "3":

FIXED(3)

represents numbers of the form

d___d.ddd

"d" = decimal digit.

If initialized with the value 1, the internal representation of a FIXED(3) variable, in the least significant word, is:

```
                                1 1 1 1 1 1
 ...  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5

 ...  0 0 0 0 0 0 1 1 1 1 1 0 1 0 0 0  = 1000 internally
```

If <fpoint> is omitted, it means that the implied decimal point is immediately to the right of the least significant digit. An omitted <fpoint> (or an <fpoint> of 0) represents numbers of the form:

d___d.

If initialized with the value 1, the internal representation of a FIXED(0) variable, in the least significant word, is:

```
                                1 1 1 1 1 1
 ...  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5

 ...  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1  = 1 internally
```

IF <fpoint> is a negative value, it indicates the number of digits to the right of the least significant digit that the implied decimal point is positioned. For example, an <fpoint> of "-3" (verbally, minus three):

FIXED(-3)

represents numbers of the form

d___d000.

A FIXED(-3) variable cannot be initialized with a value less than 1000. However, if initialized with the value 1000, the internal representation of a FIXED(-3) variable, in the least significant word, is:

```
                                1 1 1 1 1 1
 ...  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5

 ...  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1  = 1 internally
```

Scaling: "Scaling" is the act of moving the position of the implied decimal point. Internally, scaling involves multiplying or dividing an operand by a power of ten. For example: A FIXED(3) variable contains the value 1.000, internally that is represented by

1000 (decimal).

If scaled by a factor of two, to represent the value 1.00000 (i.e., the implied decimal point is moved two positions to the left), the internal representation is multiplied by 100 (10^2), resulting in

100000 (decimal).

Arithmetic is permitted among FIXED operands having different implied decimal point positions; the compiler automatically emits instructions for normalizing (i.e., matching the decimal point positions or "scaling") the operands (see "Arithmetic Expressions" for a complete explanation of fixed point scaling).

When a value is assigned to a FIXED variable, the value is scaled up or down as required to match the <fpoint> of the variable. If the value must be scaled down, then some order of precision will be lost. If, for example, the value

```
2.345F  ! FIXED(3) value
```

is stored in a FIXED(2) variable, the value is scaled down one position causing a loss of one digit of precision. In this example, the value

```
2.34F   ! FIXED(2) value
```

is stored.

The compiler has the ability to automatically generate instructions for "rounding" a FIXED operand when an assignment to a variable occurs. Rounding is enabled and disabled by the two compiler control commands ?ROUND and ?NOROUND, respectively (see "Compiler Control Commands" for a complete explanation). The default condition, ?NOROUND, causes the value to be truncated if the value must be scaled down prior to the assignment (as shown in the preceding example). Specifying the ?ROUND compiler command, causes the value to be rounded up, if appropriate, after truncation occurs. For example, specifying the compiler command

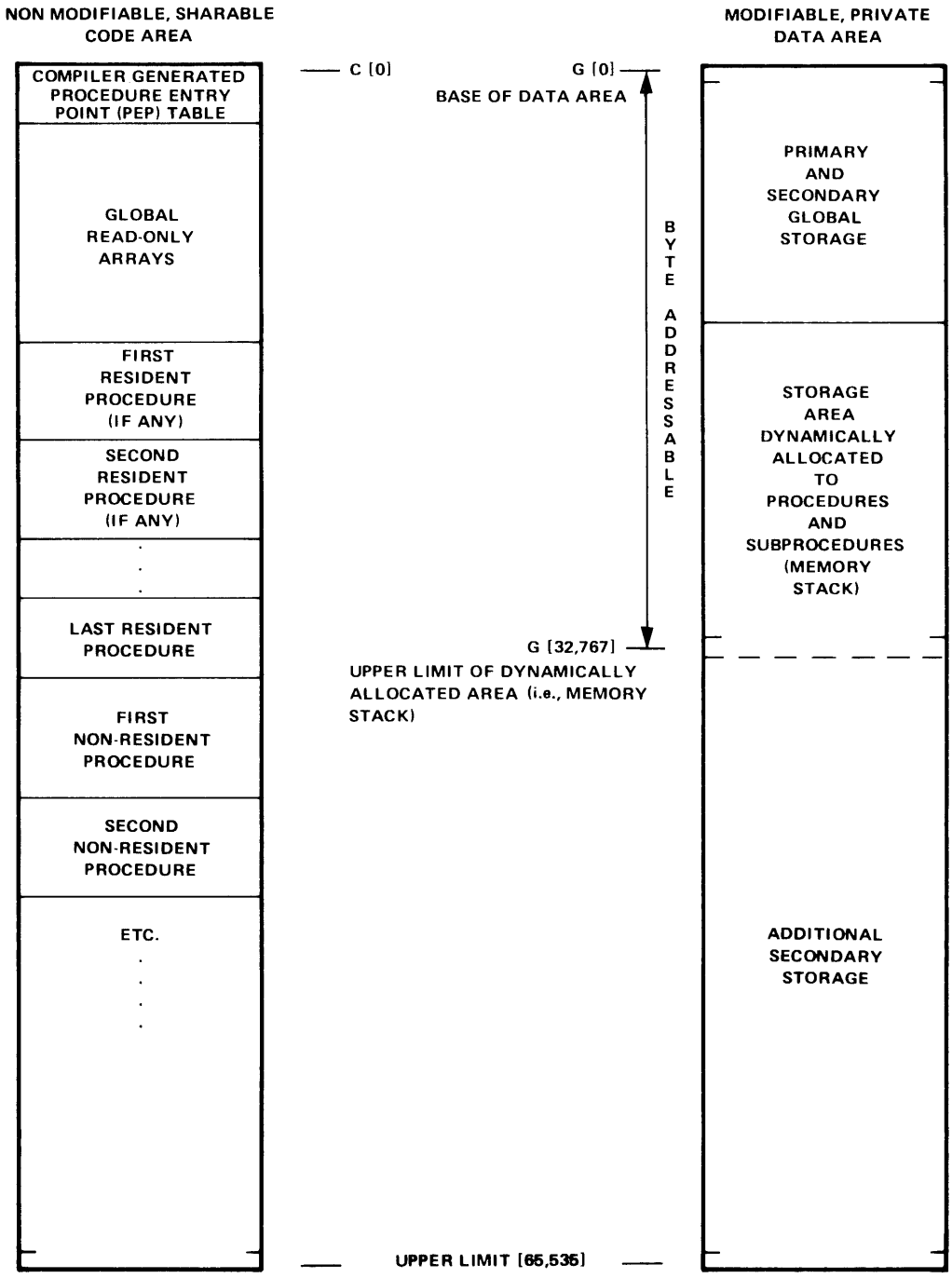
```
?ROUND
```

and assigning the value 2.345 to a FIXED(2) variable causes the value to be truncated one digit and rounded up to the value 2.35.

Additionally, two predefined functions are available in the T/TAL language for dealing with FIXED operands. One is used to move (scale) the position of the implied decimal point; the other is provides the <fpoint> value of a FIXED operand. See "Standard Functions" for a complete explanation of these functions.

A Tandem 16 object program executing in memory is physically separated into two parts: An area called the code area that contains the program's machine instructions and an area called the data area that contains the program's data variables. The code area cannot be modified (i.e., written into) while the program is executing.

OBJECT PROGRAM IN MEMORY



CODE AREA

The code area (called code because it contains instruction codes) consists of a PEP, read-only arrays, and executable instructions.

The PEP (procedure entry point table) is a list of all procedure entry points in the program. The total number of entries in the PEP is two plus the number of procedure entry points. The PEP is automatically configured by the compiler and the operating system and is referenced by the hardware when a procedure is called.

If global read-only arrays are used, they immediately follow the PEP.

Following any read-only arrays are any procedures which are designated as main memory resident. Resident procedures are physically placed in memory in the order in which they appear in the source program. The remainder of the procedures follow the main memory resident procedures. They are also physically placed in the same order as they appear in the source program.

The hardware provides word and byte addressing of constants in all of the code area. Doubleword and quadrupleword addressing of constants in the code area is provided automatically in the T/TAL language.

DATA AREA

The maximum memory data storage area allowable for any single program is 65,536 words. The first word in the data area, the base address, is designated 'G'[0] (for global address zero). Words in the data area are numbered consecutively, starting from the base, extending through 'G'[65535].

At any given moment, the data area is logically comprised of up to three separately and independently accessed areas:

1. The GLOBAL area - data variables within this area are declared at the beginning of the source program and can be accessed by any instruction in the program.

2. The LOCAL area of the currently executing procedure - data variables within this area are defined at the beginning of the procedure declaration and are accessible only by instructions in the currently executing procedure (and any subprocedures within the procedure). Local data variables that are assigned initial values when declared are initialized each time the procedure is entered.

   At any given moment, there may be a number of other LOCAL areas that are associated with procedures which are active (i.e., have been called but have not finished) but not currently executing. Any other LOCAL areas are not known to the currently executing procedure.

3. The SUBLOCAL area of the currently executing subprocedure - data variables within this area are defined at the beginning of the subprocedure declaration and are accessible only by instructions in the currently executing subprocedure. Sublocal variables that are assigned initial values when declared are initialized each time the subprocedure is entered.

   At any given moment, there may be a number of other SUBLOCAL areas that are associated with subprocedures that are active but not currently executing. Any other SUBLOCAL areas are not known to the currently executing subprocedure.

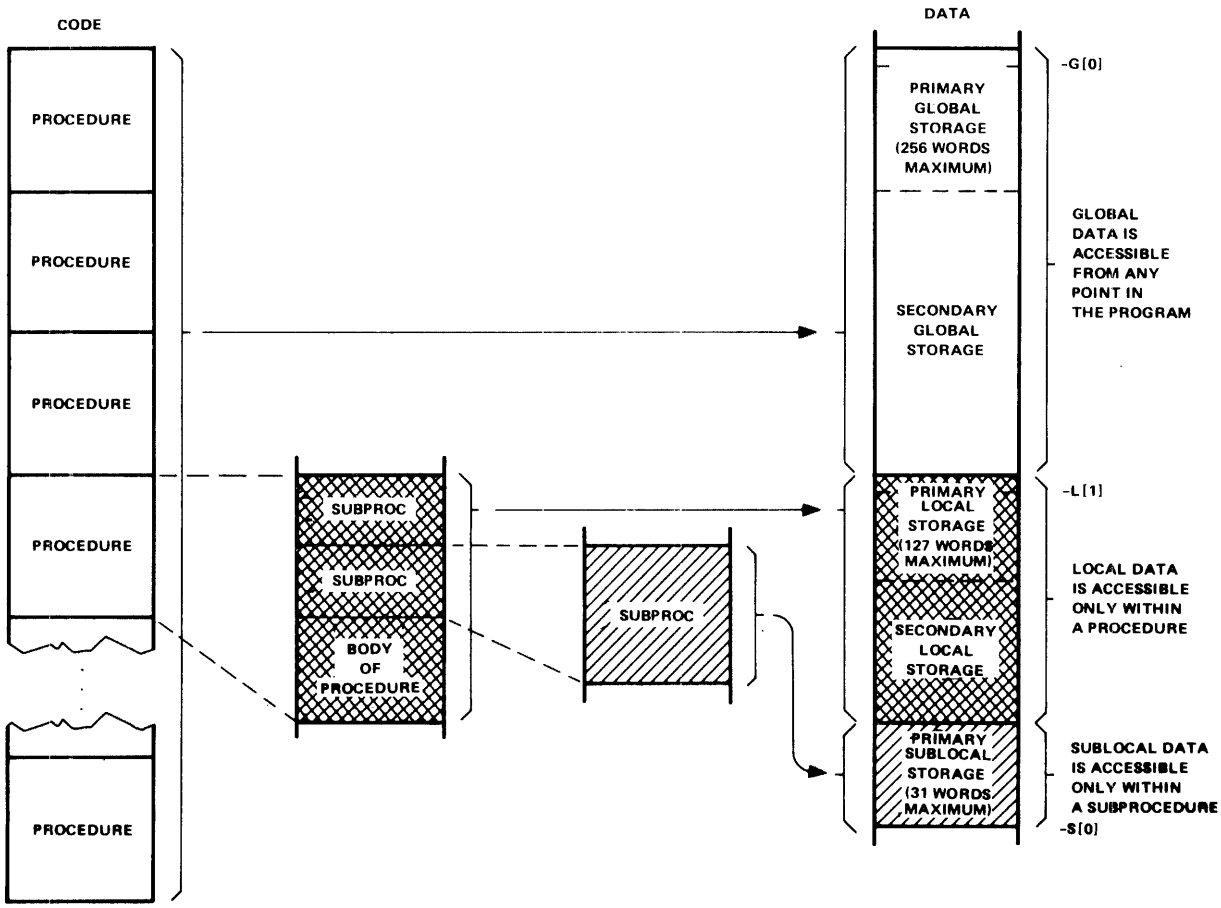
The GLOBAL and LOCAL areas are further subdivided into a directly addressable, or "primary" area, and an area called the "secondary" area. The "secondary" area can only be addressed indirectly through an address pointer or addressed by means of an element index (address pointers and element indexing are described later).

All addressing in the global area is relative to the base of the global area - 'G'. In the global data area only the first 256 words, 'G'[0:255], can be addressed directly. This "primary" area is used for simple variables, address pointers, and directly addressable arrays. The space in the primary area is allocated by the compiler as required, so it does not necessarily use the entire 256 words. The remainder of the global area, the area not "allocated" to primary storage, is addressed indirectly through address pointers located in the primary area. The "secondary" area is used for indirectly addressable arrays.

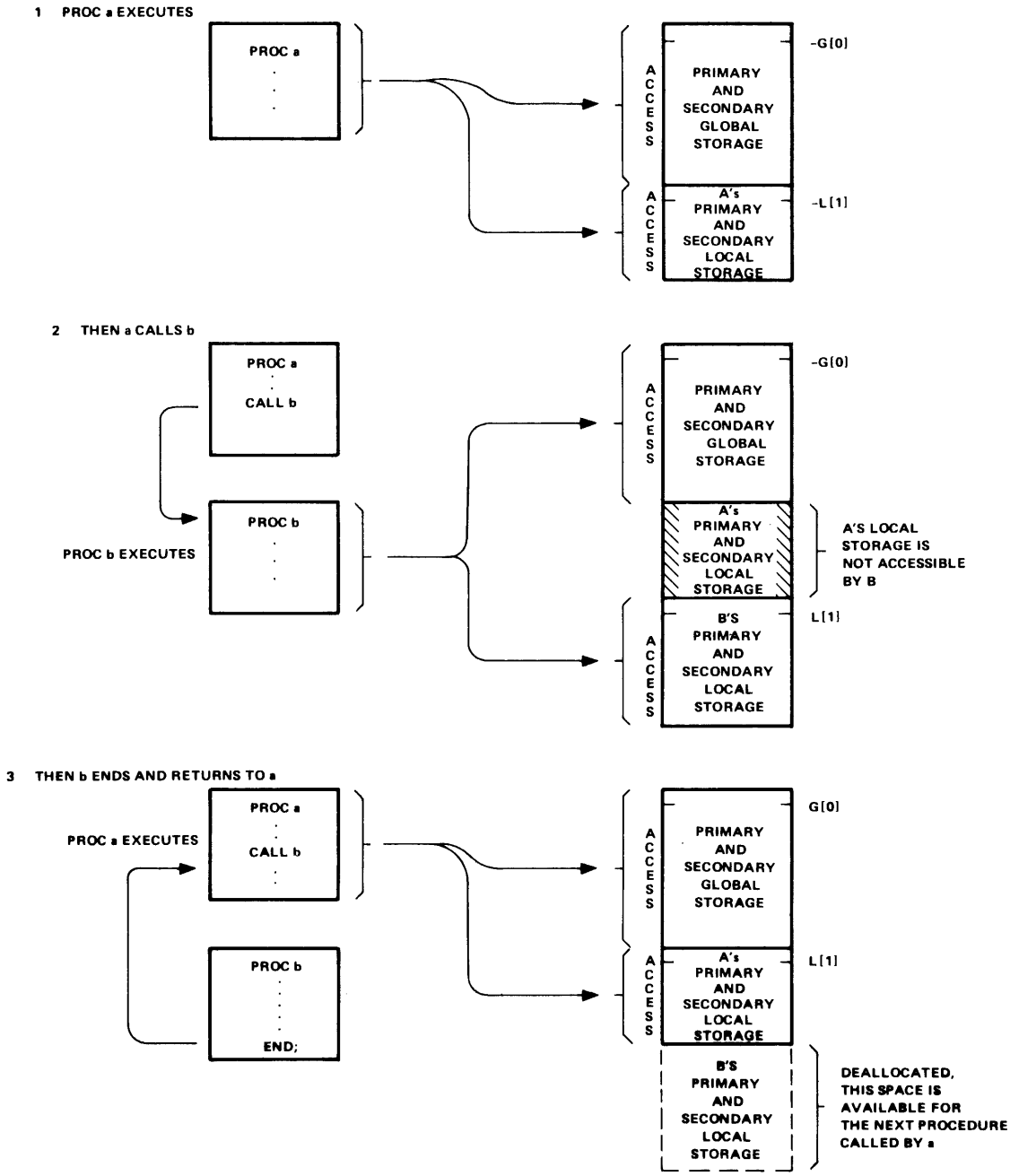
All addressing in a local area is relative to the base of that local area - 'L'. In a local data area only the first 127 words, 'L'[1:127], are directly addressable. Like the global area, space is allocated only as required. The secondary local area is addressed through pointers in the primary local area. The physical space for a local area is allocated when a procedure is called and is de-allocated when a procedure is finished.

All addressing in a sublocal area is relative to the base of that sublocal area - 'S'. (The addresses are actually a negative offset from 'S'.) The entire 31 words comprising a sublocal data area, 'S'[-30:0] are addressed directly. Any parameters passed to a subprocedure count as part of the sublocal area. Therefore, the maximum number of words allocated a given subprocedure is 31 minus the number of parameter words passed to that subprocedure. Like local areas, physical space for sublocal areas is only allocated while a subprocedure executes.

GLOBAL, LOCAL, AND SUBLOCAL DATA AREAS



DYNAMIC ALLOCATION OF LOCAL STORAGE



The storage areas that the hardware dynamically allocates to procedures and subprocedures are limited to the first 32,768 words of the program's data area (an attempt by the hardware to allocate local or sublocal storage above 'G'[32767] results in a memory stack overflow trap occurring. See the "Guardian Programming Manual" for an an explanation of traps)

The entire data area is addressable by the hardware on word, doubleword and, if a particular processor module has the Decimal Arithmetic option installed, quadrupleword boundaries. The hardware also provides byte addressing anywhere in the first 32,767 words of the data area.

When a program is run, the amount of data area that it is allocated in (main and/or virtual) memory is one page (i.e., 1024 words) past the area required for global data. The upper limit to the number of pages in a program's data area can be specified as a larger amount (up to 64 pages) when the program is run.

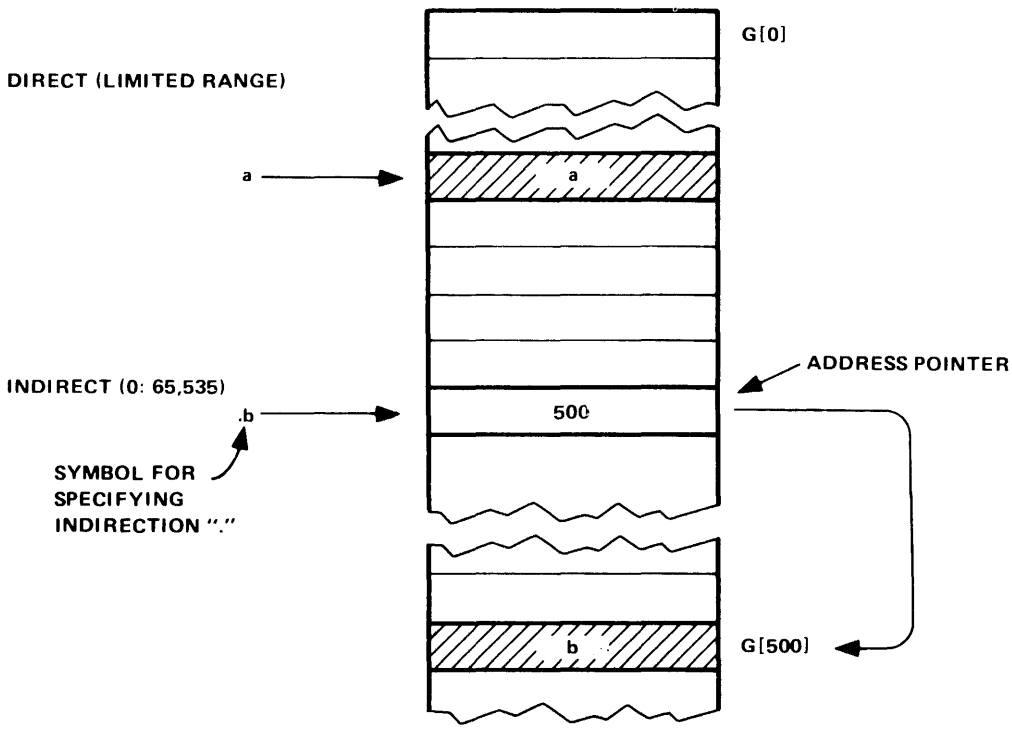
DIRECT AND INDIRECT ADDRESSING

Data elements are addressed either directly by means of an address field in a machine instruction or indirectly through an address pointer in memory. An address pointer is a word that contains the 'G'[0] relative address of another word. Direct addressing is faster because only one memory reference is made; however the range of direct addressing is limited. Indirect addressing requires two memory references (one to get the pointer contents, the second to get the actual data) but can access any location in the data area.

The addressing mode is, for the most part, implied when a data variable is declared. Direct addressing is indicated by NOT preceding the name of a variable with period ".". Indirect addressing is indicated by preceding the name of a variable with a period:

```
INT  <variable>

   is addressed directly.

INT .<variable>

  is used as indirect address pointer.
```

DIRECT VERSUS INDIRECT ADDRESSING



The 'G'[0] relative address of a variable can be obtained by preceding the name of the variable with a commercial at "@" symbol:

```
@<variable>
```

A programmer may often find that it is necessary to access a word-addressed variable (i.e., word, doubleword or quadrupleword) as a string variable or vice versa.

To access a word-addressed variable as a string variable, the programmer must declare a STRING pointer and initialize the pointer with the byte address of the variable. Initialization is accomplished by shifting the 'G'[0] relative word address of a variable left one position (see "Pointer Variables" and "Bit Shift" for a full explanation):

```
STRING .str^pointer := @word^variable '<<' 1; ! logical left shift.
```

To access a STRING variable as a word-addressed variable, a word address must be generated. This is accomplished by declaring an INT (or INT(32) or FIXED) pointer and initializing the pointer with the string address shifted right one position:

```
INT .word^pointer := @str^variable '>>' 1;  ! logical right shift.
```

To access a word-addressed variable of one <type> as a word-addressed variable of another <type>, no address conversion is needed.

ELEMENT INDEXING

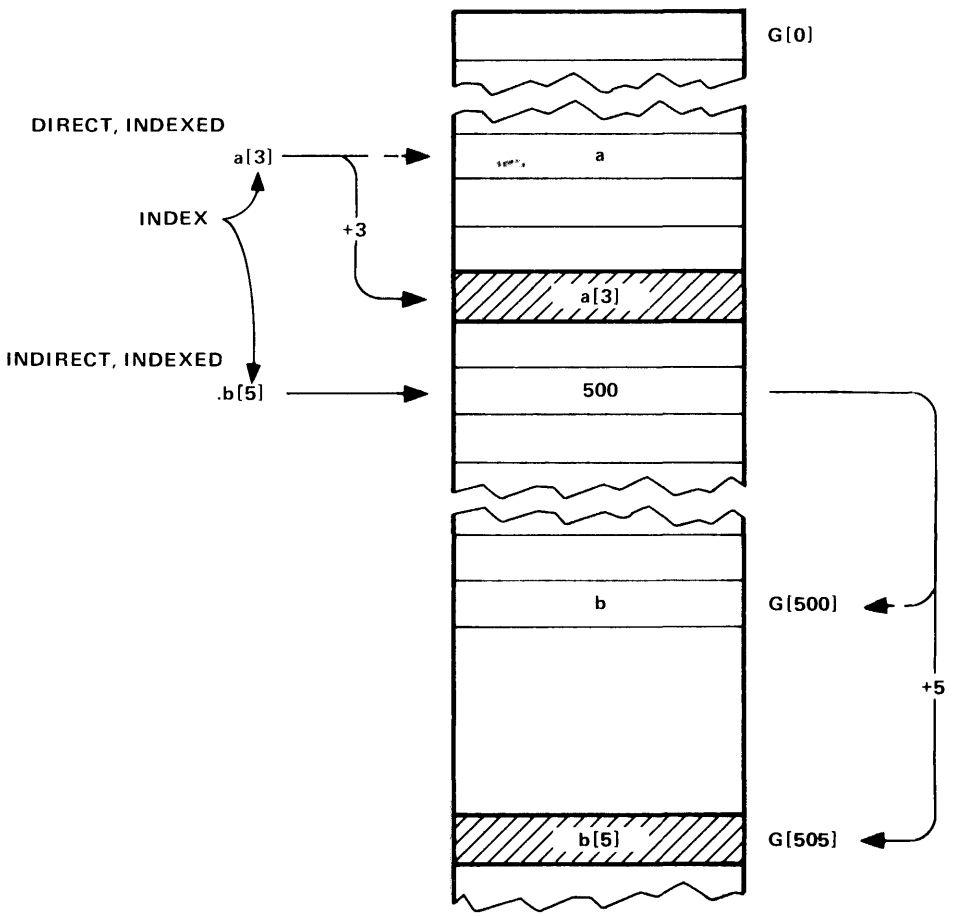
Data element indexing is provided for both directly and indirectly addressed variables. In the language, indexing is accomplished by appending an indexing subscript to the name of a variable:

<variable> [<index>]

<index> indicates an element offset from the <variable>.

The range of indexing subscripts is [0] through [%177777]. This provides direct access to any element in the data area.

ELEMENT INDEXING



Indexing in the hardware is performed using one of the index registers; the address of a variable is determined, then the indexing value is added to provide the address of the desired element (post indexing).

Identifiers are symbols used to name elements (such as variables, labels, procedures, etc.) in a T/TAL program. They consist of letters and numbers, and are assigned uses when declared. There is no implicit data type assumed for identifiers.

An identifier always starts with a letter or a circumflex symbol "^" and contains from 1 to 31 contiguous characters (letters, digits, and circumflexes). Lower case characters are allowed, but are considered the same as uppercase when processed by the compiler.

The following are valid identifiers:

```
al

number^of^bytes

TANDEM

z1234567890^31^^

^
```

The following are invalid:

```
labc
```

is invalid because it starts with a number.

```
ab%99
```

is invalid because it contains "%".

An identifier can be known to one part of a program, but not to another:

* Identifiers assigned globally are known throughout the program
* Identifiers assigned locally are known only in the procedure where declared
* Identifiers assigned sublocally are known only in the subprocedure where declared

The same identifier can be assigned at the global, local, and sublocal levels. The hierarchy of identifiers is illustrated in the following example:

```
INT a; ! global declaration

PROC p1;
  BEGIN
     .
    INT a; ! local declaration
    SUBPROC sp1;
      BEGIN
        INT a; ! sublocal declaration
        a := 1;
          .
           uses the identifier declared sublocally
          .
      END;
    SUBPROC sp2;
      BEGIN
        a := 1;
          .
           uses the identifier declared locally
          .
      END;
    a := 1;
      .
       uses the identifier declared locally
      .
  END;

PROC p2;
  BEGIN
    a := 1;
      .
       uses the identifier declared globally
      .
  END;
```

RESERVED SYMBOLS

The T/TAL reserved symbols cannot be used as identifiers. Additionally, an external procedure name such as that of a Guardian Procedure cannot be used as an identifier if the procedure is to be called in the program.

List of Reserved Symbols

AND	END	LOR	STACK
ASSERT	ENTRY	MAIN	STORE
BEGIN	EXTERNAL	NOT	STRING
BY	FIXED	OF	STRUCT
CALL	FOR	OR	SUBPROC
CALLABLE	FORWARD	OTHERWISE	THEN
CASE	GOTO	PRIV	TO
CODE	IF	PROC	UNTIL
DEFINE	INT	REAL	USE
DO	INTERRUPT	RESIDENT	VARIABLE
DOWNTO	LABEL	RETURN	WHILE
DROP	LAND	RSCAN	XOR
ELSE	LITERAL	SCAN	

Constants are literal values that stand for themselves. Constants are used when initializing variables, declaring literals, in arithmetic and conditional expressions; anywhere a number can stand for itself.

There are four forms of constants: integer, doubleword integer, string, and fixed.

INTEGER CONSTANTS

Integer constants represent signed or unsigned 16-bit quantities (i.e., INT). An integer constant can be used anywhere a string constant is permitted.

There are two representations for integer constants: decimal integers and based integers.

Decimal integer constants represent signed or unsigned 16-bit quantities (i.e., INT). A decimal integer constant consists of an optional plus or minus sign followed by any of the decimal digits 0 through 9. The range of decimal integer constants is

-32,767 to +32,767.

Integer values greater than 32,767 decimal must be represented using based integer constants.

Note: When using an integer constant in a byte move operation, the constant should be surrounded by brackets "[...]" if a one-byte value is desired. If an integer constant value is not surrounded by brackets, it is treated as two bytes in a byte move operation.

Some examples:

13579

is a decimal integer constant.

-15791

is a signed decimal integer constant.

Based integer constants (ie, base 2 or base 8) consist of a % percent sign, followed by the character "B" if base 2, followed by the digits legal for the designated base:

%100000

is a base 8 (octal) integer constant (equivalent to 32,768 decimal).

%B1010111

is a base 2 (binary) integer constant.

An integer constant can also be represented by a constant expression. A constant expression is an arithmetic expression containing constants and literals that can be evaluated to a constant value;

```
3 * 5 + 12 / 4
```

evaluates to the constant value <18>.

```
LITERAL six = 6;                    ! LITERAL declaration

30 / six
```

evaluates to the constant value <5>.

DOUBLEWORD INTEGER CONSTANTS

Doubleword integer constants represent signed 32-bit quantities (i.e., INT(32)).

There are two representations for doubleword integer constants: decimal integers and based integers.

There are two representations for integer constants: decimal integers

Doubleword decimal integer constants are represented by appending the character "D" to a digit string consisting of a maximum of ten of the digits 0 through 9. The range of doubleword decimal integer constants is

-2,147,483,647 to +2,147,483,647.

An example:

```
14769D
```

is a 32 bit integer.

A doubleword based integer constant is indicated by appending the character "D" to a based constant:

```
%1707254361D
```

is a base 8 (octal) 32-bit integer constant.

```
%B00010010110001000101000100 1D
```

is a base 2 (binary)32-bit integer constant.

Like integer constants, doubleword integer constants can be represented by a constant expression.

STRING CONSTANTS

A string constant is a sequence of one or more ASCII characters bounded by quotation marks. A one- or two-character string constant can be used wherever an integer constant is permitted.

Lowercase characters are not upshifted in strings. To avoid confusion with the string terminator, a quote within a character string is represented by a pair of quotes.

Some examples:

```
"ABCDEFG12345"
```

is a string constant

```
"Tandem Computers Incorporated"
```

illustrates that lower case characters are not upshifted.

```
""""
```

is a string constant consisting of a single quote.

A string constant must be wholly contained within one line of text. (However, a constant list can be used to represent strings that extend over more than one line.)

FIXED CONSTANTS

Fixed constants represent signed 64-bit fixed decimal quantities.

There are two representations for fixed constants: decimal and based.

A fixed decimal constant consists of an optional plus or minus sign followed by up to 19 of the decimal digits 0 through 9 followed by an "F" (for fixed). A fixed decimal constant may have a fraction as indicated by the position of a decimal point. The range of fixed decimal constants is

-9,223,372,036,854,775,807 to +9,223,372,036,854,775,807.

Some examples:

```
1200.09F
0.1234567F
0.5F
2398409848939873494F
-10.09F
```

are all fixed decimal constants.

Fixed based constants (ie, base 2 or base 8) consist of a % percent sign, followed by the character "B" if base 2, followed by the digits legal for the designated base followed by an "F":

```
%765235512F
```

is a base 8 (octal) fixed constant

```
%B10101110101011010110F
```

is a base 2 (binary) fixed constant.

Like integer constants, fixed constants can be represented by a constant expression.

CONSTANT LISTS

A group of constants, called a constant list, can be used wherever a multiple-element constant is needed.

The general form of a constant list is:

```
"[" <constant> , ... "]"
 -  ----------       -

where

   <constant> is an integer, double integer, string, or fixed
   constant

example

   [ 1,2,3,4,5,6,7,8 ]              ! integer constant list.
```

An example of an integer constant list:

```
[ -32131, %117, 32, 64, 128, %B01010111 ]
```

A constant list composed of a number strings is treated as one contiguous string:

```
[ "A",
  "BCD",
  ".......",
  "Z" ]
```

is the same as

```
"ABCD......Z"
```

Integer and string constants can be intermixed in a constant list:

```
[ "abcdef" , 1, 2, 3, "XYZ1", %120 ]
```

REPETITION FACTORS

Repetition factors can be used to represent a list containing a recurring integer or string constant. The general form of repetition factors is:

```
<repetition factor> * "[" <constant> "]"
------------------- -  -  ----------  -

where

   <repetition factor> is an integer constant
```

Some examples of using repetition factors to represent an integer constant list:

```
10 * [ 0 ]    is equivalent to

[ 0,0,0,0,0,0,0,0,0,0 ]


[ 3 * [ 2 * [ 1 ], 2 * [ 0 ] ] ]    is equivalent to

[ 1,1,0,0,1,1,0,0,1,1,0,0 ]
```

Some examples of using repetition factors to represent a string constant list:

```
10 * [ " " ]    is equivalent to

[ "          " ]


[ 2 * [  3 * [ "ab" ], 2 * [ "xyz" ] ] ]    is equivalent to

[ "ab","ab","ab","xyz","xyz","ab","ab","ab","xyz","xyz" ]
```

Constants using repetition factors as well as constants not using repetition factors can be intermixed:

```
[ 1,2,3, 3 * [ "a" ], "B", "C", 2 * [ 0 ] ]    is equivalent to

[ 1,2,3,"a","a","a","B","C",0,0 ]
```

A variable is a symbolic representation of an element or group of elements in a program's data area. A variable can be used in an expression to produce a value. The contents of a variable can be changed through use of an assignment statement.

Data Types

Four types of data variables are defined: single word integer, double word integer, byte, and four word fixed point:

* Single word integer variables are declared as <type> INT.

* Double word integer variables are declared as <type> INT(32).

* Byte variables are declared as <type> STRING.

* Four word fixed point variables are declared as <type> FIXED. Associated with a fixed point variable declaration is an optional <fpoint> specifier:

  FIXED [(<fpoint>)]
  -----

  where

  <fpoint> specifier gives the number of positions that the implied decimal point is located to the left (<fpoint> > 0) or to the right (<fpoint> <= 0) of the least significant digit. If <fpoint> is omitted, 0 is implied. <fpoint> is represented by an integer constant. The range of <fpoint> is -19 to + 19

  examples

  FIXED(3)

  represents decimal numbers of the form

  d___d.ddd

  "d" is a decimal digit.

  FIXED

  represents decimal numbers of the form

  d___d.

  FIXED(-3)

  represents decimal numbers of the form

  d___d000.

All defined data types can be either simple variables (containing one element), array variables containing more than one element, or pointer variables (containing an address of another data element).

Initialization

Any data variable can be initialized when declared. Generally, the constant used to initialize a variable must match the <type> of the variable being declared. However, string constants can be used to initialize any type data variable. Additionally, a string variable can be initialized with an integer constant that can be represented in eight bits.

Initialized global variables have their initial values when the program starts executing. local and sublocal variables that have initial values are initialized each time the procedure or subprocedure is entered.

Address Equivalencing

Address equivalencing is provided so that a memory location can be represented by more than one variable (and more than one data type).

A simple variable represents a one element data item. A simple variable is declared by assigning a data type to an identifier.

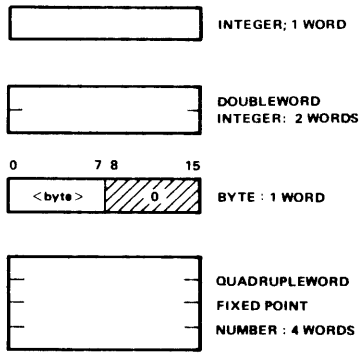
The general declaration for a simple variable is:

```
<type> { <name> [ := <initialization> ] } , ... ;
------   ------                                 -

where

   <type> is { INT                    }
             { INT(32)                }
             { STRING                 }
             { FIXED [ ( <fpoint> ) ] }

   <name> is an identifier assigned to the simple variable

   <initialization> is a constant or, if declared locally or
   sublocally, can be an arithmetic expression

   more than one simple variable of the same <type> can be
   specified and initialized per declaration (separated by
   commas ",")

examples

   INT simple^variable;
   INT(32) intial^variable := 99D;
   STRING var1, var2 := 0, var3;
   FIXED(3) fvar := 1234.456F;
```

Simple variables are allocated space in the next available primary area as they are declared. The space allocated depends on the data type:

MEMORY ALLOCATION FOR SIMPLE VARIABLES

```
INT simple^int;

INT(32) simple^dbl;

STRING simple^string;

FIXED simple^fixed;
```



Note that when a simple STRING variable is declared, the single STRING element occupies bits 0 through 7 of the word (bits 8 through 15 are set to zeros).

INITIALIZING SIMPLE VARIABLES

When an simple variable is initialized, it takes the value of the initializing constant or expression.

* Simple INT variables are initialized with integer constants and, if desired and declared locally or sublocally, expressions whose results can be represented within 16 bits.

* Simple INT(32) variables are initialized with double integer constants, and, if desired and declared locally or sublocally, arithmetic expressions that produce double integer values.

* Simple STRING variables, for purposes of initialization, are treated as INT variables with one exception. If an integer constant or an integer expression is used for initialization, only the least significant eight bits of the initializing value are used (the most significant eight bits are lost).

* Simple FIXED variables are initialized with fixed constants and, if desired and declared locally or sublocally, arithmetic expressions that produce FIXED values.

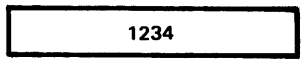
* Any <type> simple variable can be initialized with a string constant. The characters comprising the string constant are left justified in the variable being initialized. If the number of characters in the string constant does not match the number of characters that the variable can represent, the trailing (or right-side) character position(s) are set to zero(s). For example, if an INT is initialized with a single string constant, the value of the constant is placed in the left half of the INT variable and the right half is set to zero.

  If the number of characters in the string constant is greater than that representable in the variable, only the first (left side) representable characters are used. The compiler issues a warning message.

  Note that STRING initialization is also treated as INT when using string constants.

EXAMPLES OF VALID INITIALIZATION AND CORRESPONDING MEMORY ALLOCATION FOR SIMPLE VARIABLES

```
INT simple^int := 2 * 617;
```



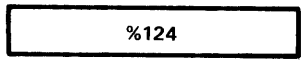

```
INT(32) simple^dbl := 987654D;
```


```
STRING simple^string := "T";
```


```
FIXED(3) simple^fixed := 3065.001F;
```

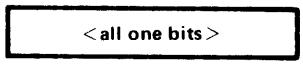

```
INT is^string := "AB";
```


```
INT octal^value := %124;
```

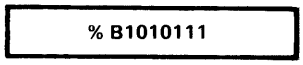



```
STRING s^octal^value := %124;
```


```
INT minus^one := -1;
```



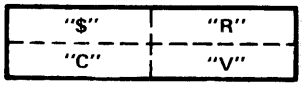

```
INT binary^value := %B1010111;
```



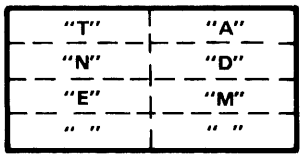

```
INT one^byte := "T";
```


```
INT two^bytes := " T";
```


```
INT(32) four^bytes := "$RCV";
```




```
FIXED eight^bytes := "TANDEM  ";
```

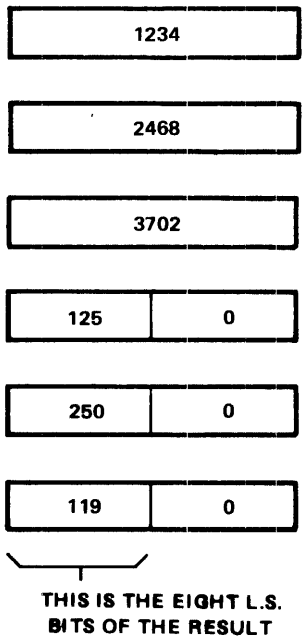


-->

EXAMPLES OF VALID INITIALIZATION (cont'd)

Initialization expressions containing variables are permitted locally and sublocally:

```
INT local^int := 1234;

INT lcl^int2 := local^int * 2;

INT lcl3 := local^int + lcl^int2;

STRING str1 := 125;

STRING str2 := str1 * 2;

STRING str3 := str1 + str2;
```



Examples of invalid initialization:

```
INT global^int := 2 * vary;
```

is invalid if declared globally.

```
INT too^long := 765D;
```

is not an integer value.

```
INT(32) too^many := "12345";
```

cannot be represented in 32 bits.

```
INT(32) not^double := -1;
```

is not a double integer constant.

```
INT too^big := 64000;
```

cannot be represented in 16 bits as a signed integer.

```
STRING dbl := 25D;
```

is not an integer or string constant.

An array variable represents a contiguous block of elements in a program's data area. An array can be thought of as a group of simple variables each having the same data type, all accessed through the same identifier. Individual elements of an array are accessed by appending an indexing subscript to the name of the array.

Some definitions regarding arrays:

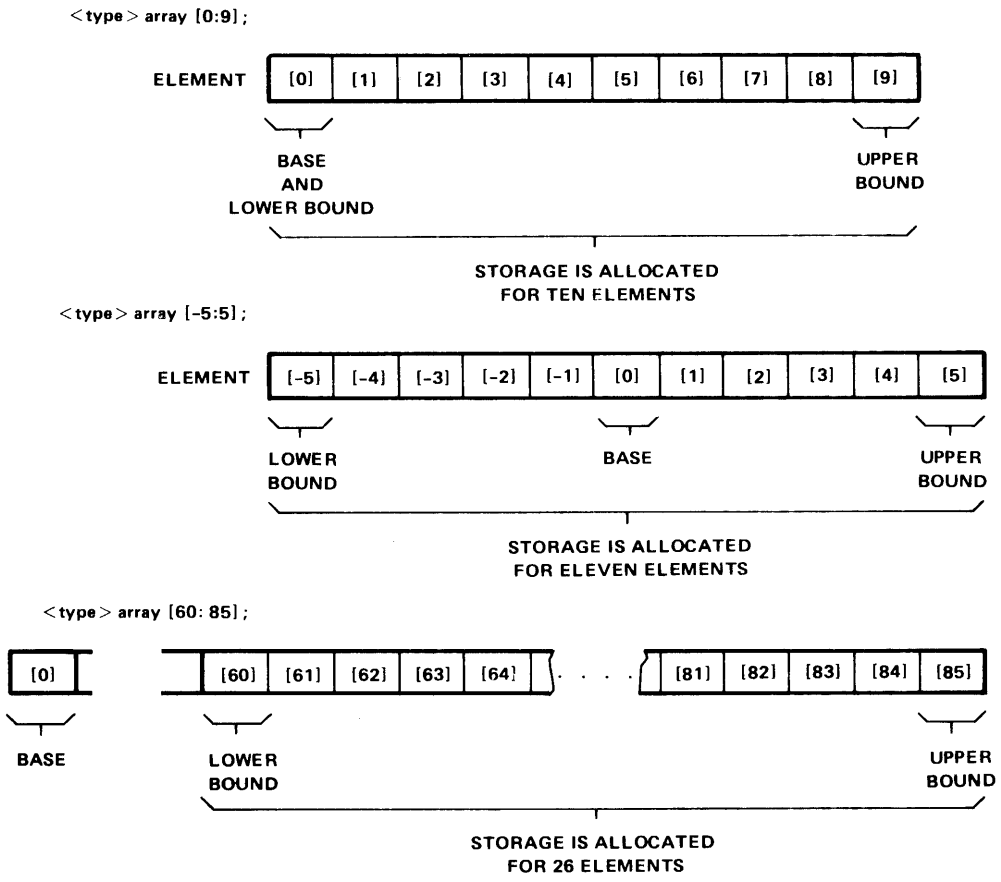
* Base - element [0] of the array.

* Lower bound - The first element of the array for which storage is allocated.

* Upper bound - The last element of the array for which storage is allocated.

The base of an array need not be within the lower and upper bounds of the array, but the base must be within the logical address space of the data area - 'G'[0:65535].

ARRAY: EXAMPLES OF BASE, LOWER BOUND, AND UPPER BOUND



Declaring Array Variables

Arrays can be either directly addressable, and allocated storage in a primary area, or indirectly addressable, and allocated storage in a secondary area. (If indirectly addressable, an address pointer is allocated in a primary area.)

An array variable is declared by assigning a data type and lower and upper bounds to an identifier. The bounds specifications must be an integer or string constants. An indirect array is declared by preceding the identifier with a period ".".

The general declaration for an array variable is:

```
<type> { [ . ] <name> "[" <lower bound> : <upper bound> "]"
       [ := <initialization> ] } , ... ;

where

   <type> is { INT                    }
             { INT(32)                }
             { STRING                 }
             { FIXED [ ( <fpoint> ) ] }

   . is the indirection symbol.  Its presence means allocate
   storage for the array in the appropriate global or local
   secondary area.  Its absence means allocate storage for the
   array in the appropriate global, local, or sublocal primary
   area

   <name> is an identifier assigned to the array

   <lower bound> is an integer constant defining the first array
   element.  The <lower bound> must be less than or equal to the
   <upper bound>

   <upper bound> is an integer constant defining the last array
   element

   <initialization> is a constant or constant list (including
   repetition factors) to be assigned as an initial value

   more than one array variable of the same <type> can be
   specified per declaration (separated by commas ",")

examples

   INT d^array [ 0:9 ];              ! direct, 10 elements.
   INT .ind^array [ 0:71 ];          ! indirect, 72 elements.
   STRING alphabet ["A":"Z"];        ! direct, 26 elements.
   STRING array1[0:9], .array2[0:9] := 10 * [ " " ], array3[0:1];
```

DIRECT VERSUS INDIRECT ARRAYS

Direct arrays are accessed faster than indirect arrays but have a limited amount of storage area available. They are typically used where a limited amount of information is accessed a relatively high number of times throughout the execution of a program. The amount of space available to indirect arrays is limited only by the total data area allocated when a program is run.

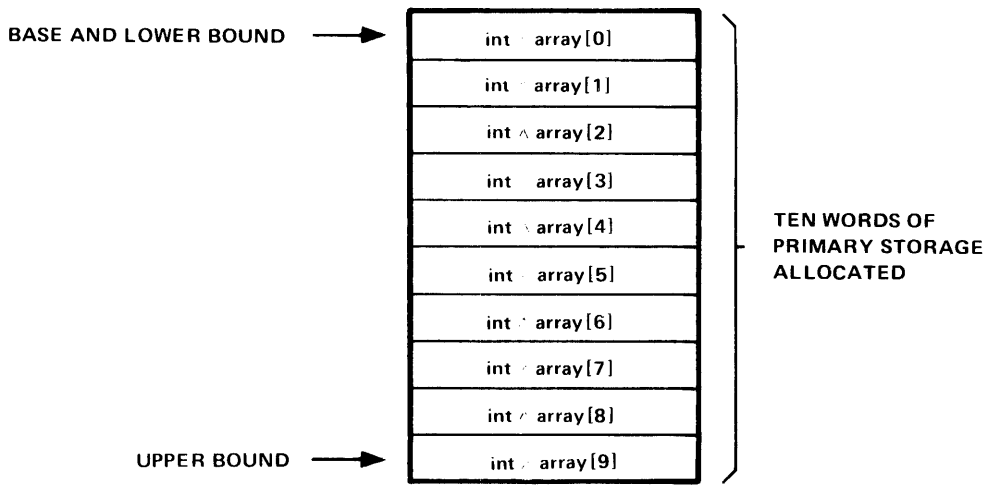
DIRECT ARRAYS

When an array is declared as directly addressable, the array is allocated in the next available primary locations. There is no address pointer associated with a direct array. The amount of space allocated to a direct array is dependent upon its data type:

```
INT arrays      = one word per element
INT(32) arrays  = two words per element
STRING arrays   = one-half word per element
FIXED arrays    = four words per element
```

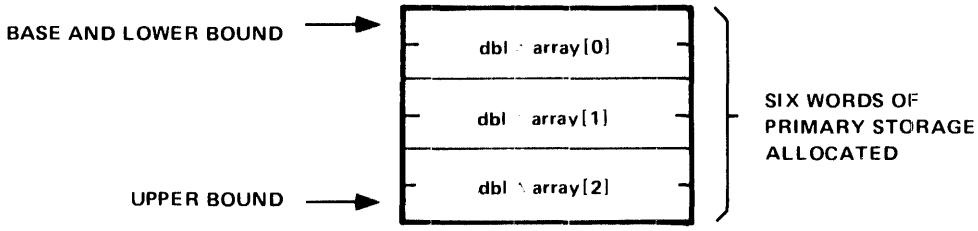
EXAMPLE OF "INT" ARRAY MEMORY ALLOCATION (ONE WORD PER ELEMENT)

```
INT int^array[0:9];
```



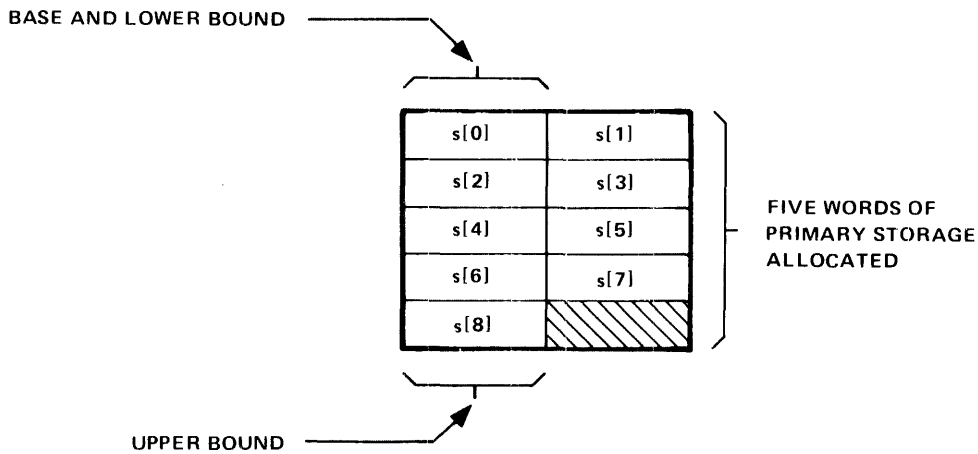
EXAMPLE OF "INT(32)"ARRAY MEMORY ALLOCATION (TWO WORDS PER ELEMENT)

```
INT(32) dbl^array[0:2];
```



EXAMPLE OF "STRING" ARRAY MEMORY ALLOCATION (ONE-HALF WORD PER ELEMENT)

```
STRING s[0:8];
```



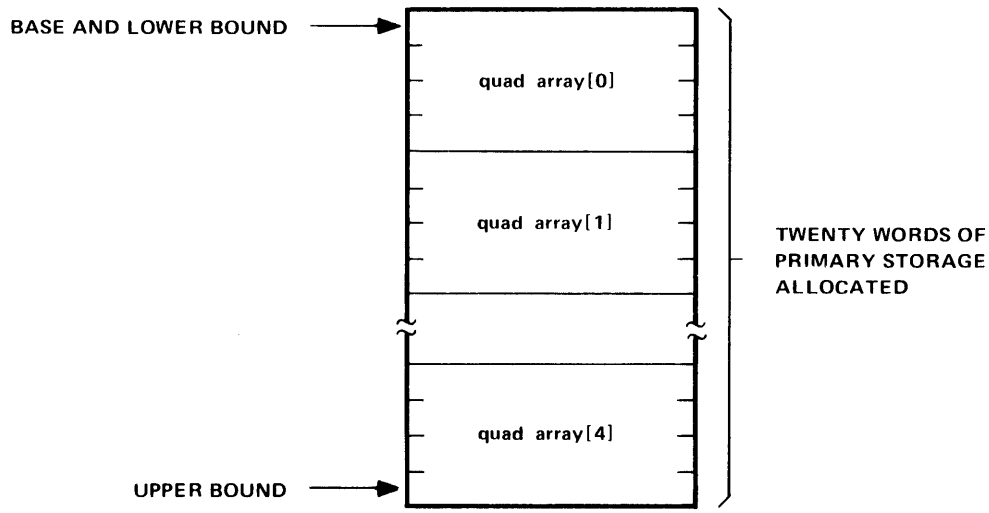
Note that if the lower bound of the array is an odd number, the first element of the array is in the right half of a word. Therefore, if the array in the preceding example were declared as

```
STRING s[ 1:8 ];
```

the preceding diagram is still valid.

EXAMPLE OF "FIXED" ARRAY MEMORY ALLOCATION (FOUR WORDS PER ELEMENT)

```
FIXED quad^array[0:4];
```



INDIRECT ARRAYS

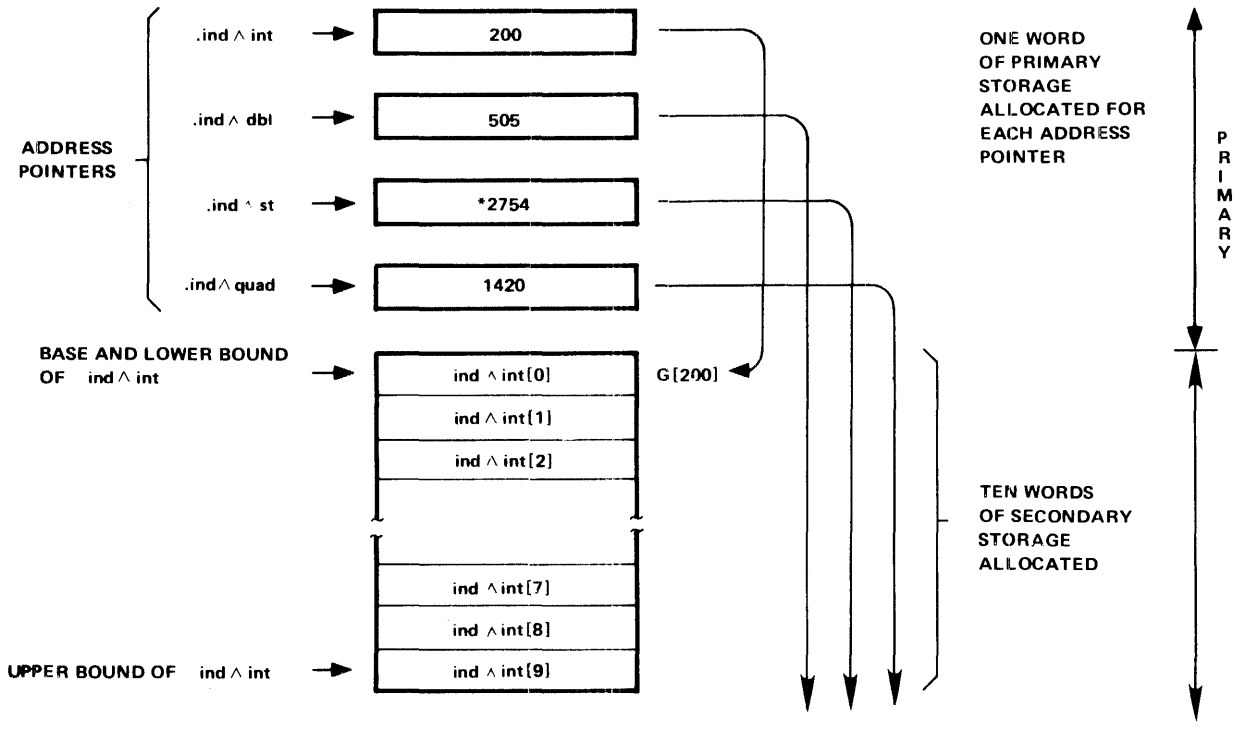
When an indirect array is declared, an address pointer to the array is allocated in a primary (directly addressable) word; the storage for the array is allocated in a secondary (indirectly addressable) area. The address pointer contains the 'G'[0] relative address of the base (i.e., element [0]) of the array.

The amount of space allocated to an indirect array is dependent upon its data type:

```
INT arrays      = a one word pointer in the primary area
                  + one word per element in the secondary area
INT(32) arrays  = a one word pointer in the primary area
                  + two words per element in the secondary area
STRING arrays   = a one word pointer in the primary area
                  + one-half word per element in the secondary area
FIXED arrays    = a one word pointer in the primary area
                  + four words per element in the secondary area
```

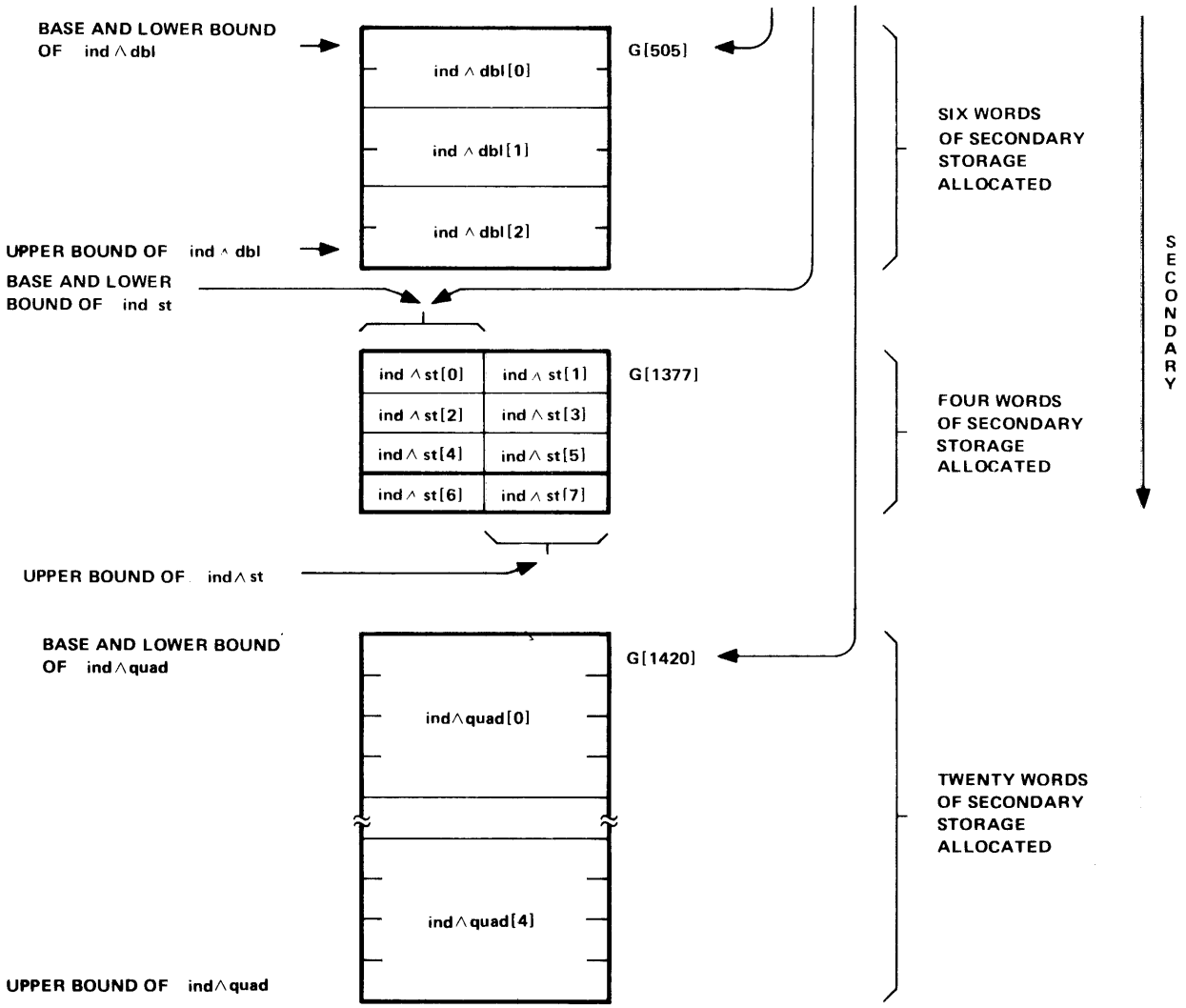
EXAMPLES OF POINTER AND STORAGE ALLOCATION FOR INDIRECT ARRAYS

```
INT .ind^int[0:9];
INT(32) .ind^dbl[0:2];
STRING .ind^st[0:7];
FIXED .ind^quad[0:4];
```



-->

EXAMPLES OF POINTER AND STORAGE ALLOCATION FOR INDIRECT ARRAYS (cont'd)



*WORD ADDRESS = $\frac{\text{BYTE ADDRESS}}{2}$

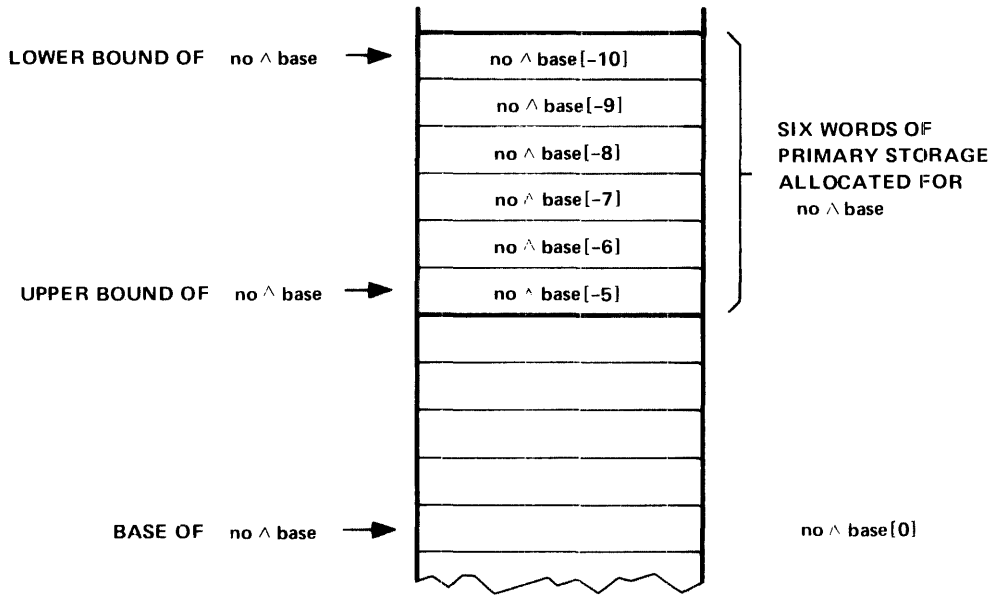
BASE ADDRESS

The base address of the array (the address used when referencing the array without a subscript) is adjusted to point to element [0] of the array (even though element [0] may not be within the bounds of the array declaration).

EXAMPLE OF ARRAY BASE NOT WITHIN BOUNDS OF ARRAY

```
INT no^base [-10:-5];
```

is a direct integer array containing six elements whose base is not within bounds.

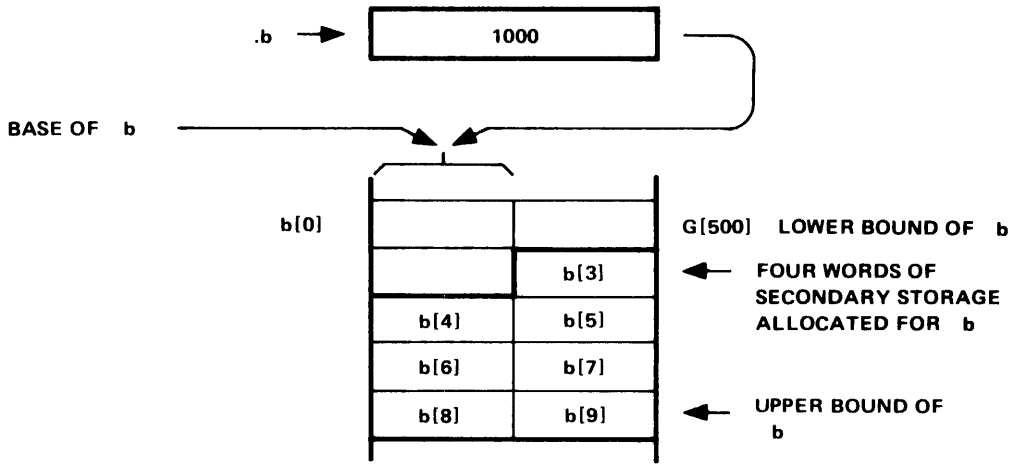


Any reference to "no^base" element [0] will access a location outside of the area allocated to "no^base".

EXAMPLE OF ARRAY BASE NOT WITHIN BOUNDS OF ARRAY

```
STRING .b[3:9];
```

is an indirect byte array containing seven elements whose base is not within bounds.



Any reference to "b" element [0] will access a location outside of the area allocated to "b".

INITIALIZING ARRAYS

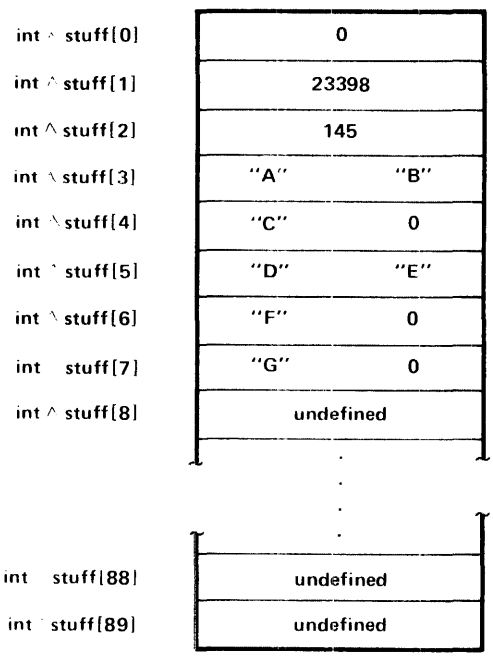
An array can be initialized with a numerical and/or string constant or constant list when declared. If initialization is with a numerical constant and the array is not an INT array, the constant must be of the proper type: STRING arrays are initialized only with integer constants; INT(32) arrays are initialized only with double integer constants; FIXED arrays are initialized only with fixed constants. INT arrays can be initialized with integer, double integer, or fixed numerical values (double integer initialization occupies two words, fixed initialization occupies four words). The initialization characteristics of arrays, as far as numerical initialization is concerned, is identical to that of simple variables.

Arrays of any data type can be initialized with string constants. A contiguous block of characters that comprises a string constant can initialize multiple elements of an array. A string constant occupies one-half word; string constant initialization always begins on an element boundary.

Initialization always begins with the lower bound of an array. Each constant in a constant list begins on an element boundary (i.e., word for INT, double word for INT(32), half-word for STRING, and quadruple word for FIXED).

EXAMPLES OF ARRAY INITIALIZATION

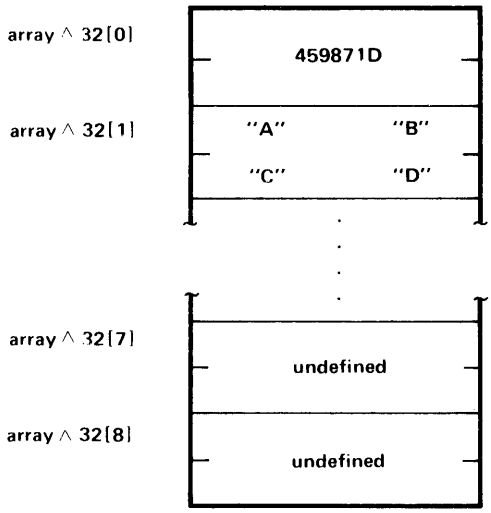
```
INT int^stuff [0:89] := [ 0, 23398, 145, "ABC", "DEF", "G" ];
```



-->

EXAMPLES OF ARRAY INITIALIZATION (cont'd)

```
INT(32) array^32 [0:8] := [ 459871D, "ABCD" ];
```




```
STRING bytes [-5:17] := [ "Tandem",
                          "Computers",
                          "Inc." ];
```

bytes [-5]		"T"
bytes [-4:-3]	"a"	"n"
bytes [-2:-1]	"d"	"e"
bytes [0:1]	"m"	"C"
bytes [2:3]	"o"	"m"
bytes [4:5]	"p"	"u"
bytes [6:7]	"t"	"e"
bytes [8:9]	"r"	"s"
bytes [10:11]	"I"	"n"
bytes [12:13]	"c"	"."
bytes [14:15]	undef.	undef.
bytes [16:17]	undef.	undef.

undef = UNDEFINED

-->

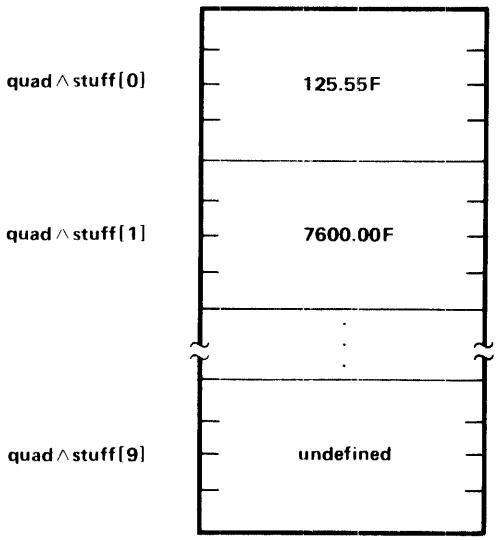
EXAMPLES OF ARRAY INITIALIZATION (cont'd)

```
STRING numbers [0:8] : = [ 70, 65, 73, 76, %40, %123 , %117,
                           %106, %124 ];
```

numbers [0:1]	70	65
numbers [2:3]	73	76
numbers [4:5]	%40	%123
numbers [6:7]	%117	%106
numbers [8:9]	%124	0

```
FIXED(2) quad^stuff[0:9] := [ 125.55F, 7600F ];
```

Note that the constant "7600F is scaled by a factor of two to match the <fpoint> of "fixed^stuff".



By using repetition factors large arrays can be initialized with recurring constants without having to actually write each part:

```
STRING blanks [0:79] := 80 * [ " " ];
```

is a string containing 80 blanks.

```
STRING heading [0:62] := [ 4 * [ "ITEM   COST   " ], "  TOTAL" ];
STRING undline [0:62] := [ 8 * [  4 * [ "-" ],"   " ], "  -----" ];
```

is equivalent to

```
STRING heading [0:62] :=
 "ITEM   COST   ITEM   COST   ITEM   COST   ITEM   COST     TOTAL";

STRING undline [0:62] :=
 "----   ----   ----   ----   ----   ----   ----   ----     -----";
```

```
INT zeros [ 0:31 ] := 32 * [ 0 ];
```

is an array containing 32 zeros.

Declaring Read-Only Arrays

Arrays containing data that will only be read can be embedded in the code area, thereby saving space in the data area. Read-only arrays must be initialized when declared.

The general form for read-only arrays is:

```
<type> { <name> [ "[" <lower bound> : <upper bound> "]" ] = 'P'
------   ------                                        - ---
         := <initialization> } , ... ;
         -- ---------------       -

where

   <type> is { INT                    }
             { INT(32)                }
             { STRING                 }
             { FIXED [ ( <fpoint> ) ] }

   <name> is an identifier assigned to the read-only array

   <lower bound> is an integer constant defining the first array
   element

   <upper bound> is an integer constant defining the last array
   element

   <initialization> is an initializing constant or constant list
   (including repetition factors)

   more than one array variable of the same <type> can be
   specified per declaration (separated by commas ",")

examples

   STRING message = 'P' := "** LOAD MAG TAPE #00144";
   INT a = 'P' := 1234; ! constant
```

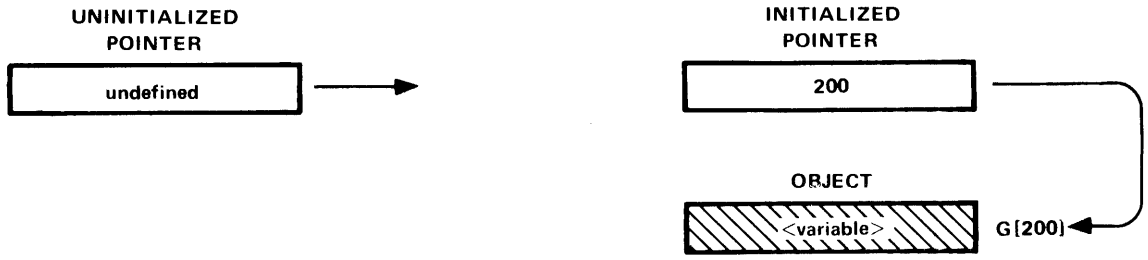
* If the <lower bound> and <upper bound> are omitted, a <lower bound> of [0] is implied, and <upper bound> of number of initialized elements minus one is implied.

* The identifier representing a read-only array cannot be passed as a variable to a procedure or subprocedure. The data in the read-only array must first be moved to an array in the data area.

* If any procedures are designated main memory resident, any global read-only arrays will also be main memory resident.

A pointer variable represents a word in memory whose contents are used as the address of an element in the data area.

POINTER VARIABLE



To use a pointer variable, it must first be initialized with the 'G' [0] relative address of a data element. The pointer variable is then used in the same manner as a simple variable in an expression or statement. The data element accessed through the pointer is treated as the data type assigned to the pointer variable.

A pointer variable is declared by preceding a typed identifier with a period ".". The general declaration for a pointer variable is:

```
<type> { . <name> [ := @<variable> "[" <index> "]" ] } , ... ;
------   - ------                                          -

where

   <type> is { INT                    }
             { INT(32)                }
             { STRING                 }
             { FIXED [ ( <fpoint> ) ] }

   . is the indirection symbol

   <name> is an identifier assigned to the pointer variable

   @<variable> provides the G[0] relative address of <variable>.
   The 'G' [0] relative address of an indexed element can be
   obtained by appending an <index> value to the <variable>.

   A pointer variable can also be initialized with a an integer
   constant or, if declared locally or sublocally, an <arithmetic
   expression>

   more than one pointer variable of the same <type> can be
   specified per declaration (separated by commas ",")
                                                          -->
```

examples

```
INT .pointer := @array;
STRING .s^pointer := @bytes[3], s^ptr2;
```

Pointer variables are allocated one word, regardless of their type, in the next available primary area as they are declared.

```
INT .int^pointer;                    ! integer pointer.
INT(32) .dbl^pointer;                ! doubleword pointer.
STRING .string^ptr;                  ! string pointer.
FIXED(3) .fpointer;                  ! fixed pointer.
```

INITIALIZING POINTER VARIABLES

When a pointer variable is initialized, the address that the pointer represents is determined by the value of the initializing address, constant, or expression. The initialization value must be capable of being represented within 16 bits, otherwise the compiler issues an error message.

Examples of valid initialization and corresponding allocation:

```
INT     .b[0:3]; ! indirect INT array containing four elements.
INT(32)  c,      ! simple INT(32) variable.
        .d[0:5]; ! indirect INT(32) array containing six elements.
STRING  .e[0:9]; ! indirect STRING array containing ten elements.

INT .p1,
```

is an uninitialized INT pointer variable.

```
    .p2 := @b,
```

is initialized with the 'G'[0] relative address of the base of "b".

```
    .p3 := @c,
```

is initialized with the 'G'[0] relative address of "c". The contents of "c" are then treated as INT values when accessed through the pointer "p3".

```
    .p4 := %100000,
```

is initialized to point at 'G'[32768]. Accessing through "p4" gets to the upper half of the data area

```
    .p5 := @d;
```

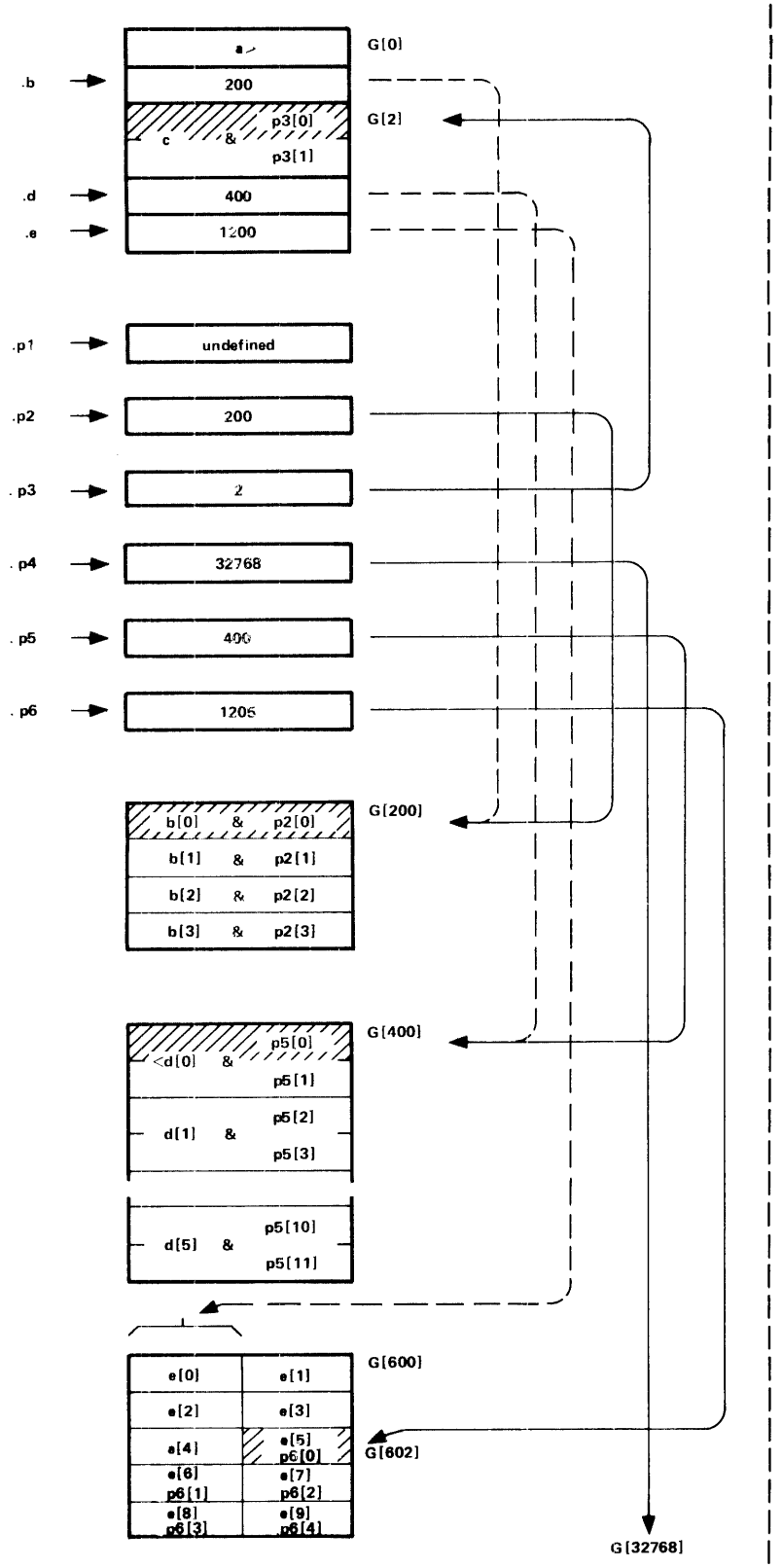
is initialized with the 'G'[0] relative address of "d". The contents of "d" are treated as INT values when accessed through the pointer "p5".

```
STRING .p6 := @e[5];
```

is a STRING pointer that is initialized with the 'G'[0] relative address of the fifth element of "e" (in STRING form). Accessing through "p6", with no subscript, gets "e[5]".

EXAMPLES OF POINTER INITIALIZATION AND ALLOCATION

```
INT    .b[0:3];
INT(32) c,
        .d[0:5];
STRING .e[0:9]

INT .p1,
    .p2 := @b,
    .p3 := @c,
    .p4 := %100000,
    .p5 := @d;

STRING
    .p6 := @e[5];
```



Dynamic Initialization of Pointer Variables

Pointer variables contents can be initialized or changed in the executable part of the program (i.e., the statements) through use of the assignment statement. Initialization is accomplished by preceding the name of the pointer with an at "@" symbol:

```
@ <pointer variable> := @<variable> [ "[" <index> "]" ]
- ----------------- -- -----------

where

   @ is the symbol for removing indirection.

   := is the assignment operator
```

```
@int^pointer := @some^array;
```

puts the 'G'[0] relative the address of "some^array" is placed in "int^pointer".

```
@pointer := @int^array[ 3 ];
```

"pointer" is initialized with 'G'[0] relative address of the third element of "int^array".

```
@string^pointer := 29;
```

points to the 30th byte in the data area.

Arithmetic with Pointer Variables

Unsigned arithmetic should always be used when dealing with the contents of pointer variables. Unsigned arithmetic is characterized by not being subject to the arithmetic overflow condition. Unsigned arithmetic is indicated by surrounding an arithmetic operator (i.e., +, -, *, /, \) by apostrophes.

For example, to increment the address in a pointer variable, unsigned add should be used:

```
@pointer := @pointer '+' 1;
```

Making a STRING Pointer Point to a Word-addressed Variable

If a STRING pointer is initialized to point to an INT, INT(32), or FIXED variable, the word address must be shifted left one position to provide a byte address.

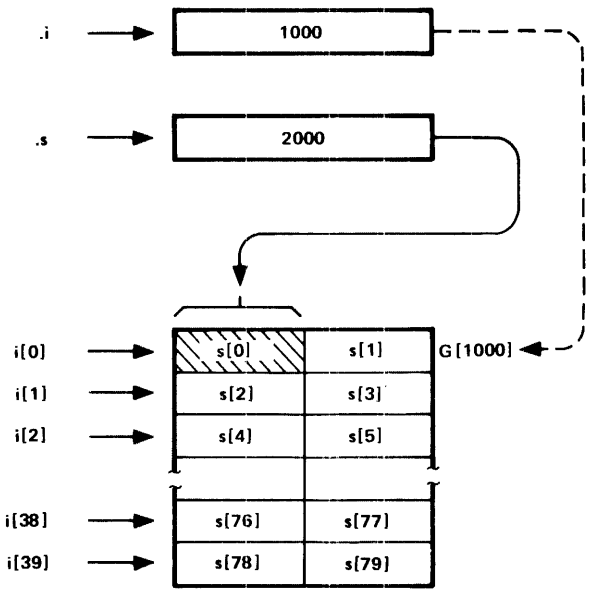
```
<string pointer> := @<word variable> [ "[" <index> "]" ] '<<' 1
```

where

'<<' means logical shift left one position (multiply by two)

EXAMPLE OF STRING POINTER TO WORD ADDRESSED VARIABLE

```
INT .i[0:39];  ! indirect INT array containing 40 elements

STRING .s := @i '<<' 1;
```



The above could be performed in the executable part of the program by using an assignment statement:

```
@s := @i '<<' 1;
```

Making a Word-address Pointer Point to a String Variable

If an INT, INT(32), or FIXED pointer is to point to a STRING variable, the byte address must be shifted right one position (divided by two) to provide a word address. Note that this action truncates the byte address down to an even number.

```
<word pointer> := @<string variable> [ "[" <index> "]" ] '>>' 1

where

   '>>' means logical right shift one position (divide by two)
```

Address equivalencing permits more than one variable to represent a given location and in doing so, permits that location to be treated as more than one data type (e.g., a declared INT can be accessed as STRING). No storage is allocated for an equivalenced variable.

The general form for address equivalencing is :

```
                                          [ "[" <index> "]" ]
<type> { [ . ] <name> = <variable> [  <word offset>  ] } , ... ;
------         ------ - ----------                              -

where

   <type> is { INT                    }
             { INT(32)                }
             { STRING                 }
             { FIXED [ ( <fpoint> ) ] }

   . is the indirection symbol.  Its presence means that the
   equivalenced variable is treated as a pointer variable.  Its
   absence means that the equivalenced variable is treated as a
   simple variable

   <variable> is either a simple, array, pointer, or another
   equivalenced variable that was previously declared

   <index> is an integer constant and is permitted only if
   <variable> is directly addressed

   <word offset> is an integer constant and is permitted with
   either directly or indirectly addressed <variables>

   more than one <equivalenced variable> of the same <type> can
   be specified per declaration (separated by commas ",")

example

   INT word = double;
```

The location indicated by <index> and <word offset> must be within the range of direct addressing for the particular data area:

for global variables the range is 'G'[0:255]

for local variables the range is 'L'[-31:127]

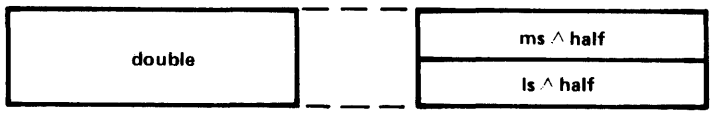
for sublocal variables the range is 'S'[-31:0]

EXAMPLE OF EQUIVALENCING ONE SIMPLE VARIABLE WITH ANOTHER

```
INT vary;
STRING st = vary;
```

EXAMPLE OF EQUIVALENCING TWO "INT" VARIABLES TO AN "INT(32)" VARIABLE

```
INT(32) double;

INT ms^half = double,
    ls^half = ms^half + 1;
```



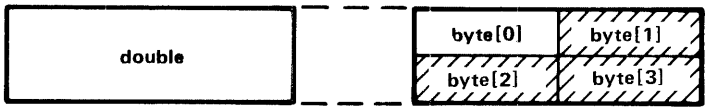
This provides a means of referencing either half of an INT(32) variable independently from the other

The above example could also be accomplished as follows:

```
INT ms^half, ls^half;

INT(32) double = ms^half;
```

EXAMPLE OF EQUIVALENCING A "STRING" VARIABLE TO AN "INT(32)" VARIABLE

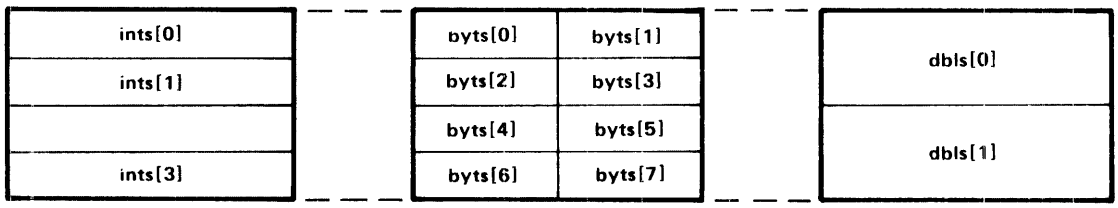
```
INT(32) double;

STRING byte = double;
```



The INT(32) variable can then be referenced as a direct STRING array consisting of four elements.

EXAMPLE USAGE OF EQUIVALENCING TO REFERENCE AN ARRAY OF ONE TYPE AS ANOTHER TYPE

```
INT ints[0:3]; ! direct array

STRING byts = ints;

INT(32) dbls = byts;
```

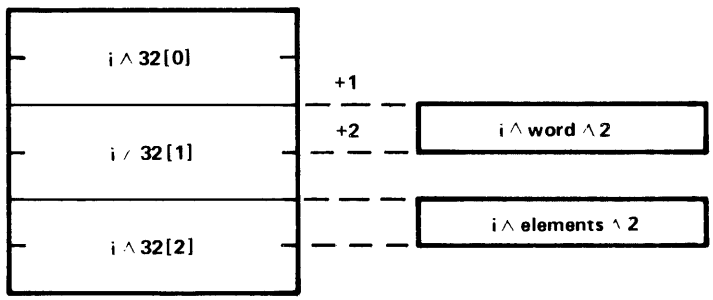


Referencing "byts" elsewhere treates "ints" as STRING values.
Referencing "dbls" elsewhere treates "ints" as INT(32) values.

An <index> or a <word offset> can be appended to the variable to equivalence to a location different than the variable. <element index> means that the number of words of offset from <variable> are related to the data type of <variable>. <word offset> means that the offset will be the number of words specified.

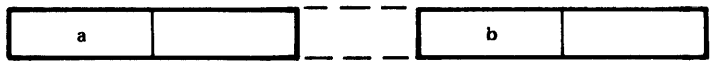
EXAMPLE OF USING AN "INDEX" WHEN EQUIVALENCING

```
INT (32) i^32[0:2]; ! direct INT(32) array consisting of three
                    ! elements.

INT i^elements^2 = i^32[2],

                    ! equivalences to two elements (four words)
                    ! from the base of "i^32".

    i^word^2 = i^32 + 2;

                    ! equivalences to two words (one element)
                    ! from the base of "i^32".
```



EXAMPLE OF ANOMALY WHEN USING AN INDEX AND EQUIVALENCING STRING VARIABLES

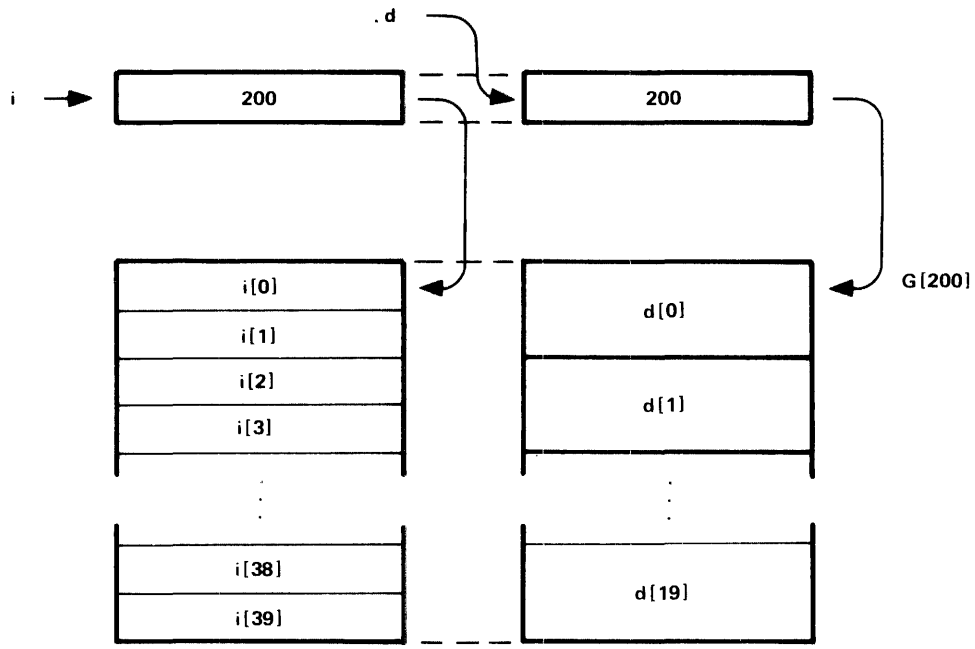
```
STRING a,
       b = a[1];
```



Equivalences to the same word and therefore the same element (both are are simple variables and simple STRING variables occupy the left-half of a word).

EXAMPLE OF EQUIVALENCING TWO WORD-ADDRESSABLE ARRAYS OF DIFFERENT DATA TYPES

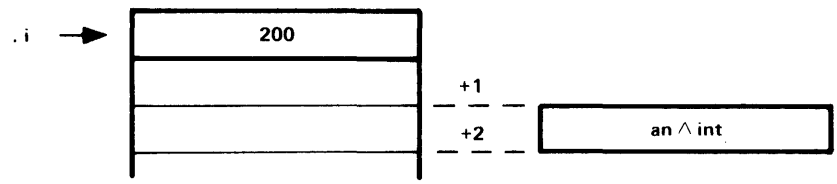
```
INT .i[0:39];  ! indirect array

INT(32) .d = i;
```



Referencing "d" elsewhere treats "i" as INT(32) values.

EXAMPLE OF USING A "WORD OFFSET" WHEN EQUIVALENCING A SIMPLE VARIABLE WITH AN INDIRECT VARIABLE

```
INT .i[0:39], ! indirect array
     an^int = i + 2;
```



Referencing "an^int" elsewhere accesses two words above the address pointer for "i".

A variable CANNOT be equivalenced to an indirect variable with an <index>:

```
INT .i[0:39]; ! indirect array

INT no^good = i[1];
```

is ILLEGAL.

One method of equivalencing to an indirect, indexed array is to assign a pointer:

```
INT(32) .dbl^array := @int^array[4];
```

A STRING array should not be equivalenced to an indirect INT or INT(32) array:

```
INT .some^array [0:79];
STRING .use^less = some^array;
```

Because no new pointer is assigned, the address referenced by "use^less" is a word address, not a byte address.

In this case it is useful to equivalence the INT array by assigning a pointer:

```
STRING .use^ful := @ some^array '<<' 1;
```

The following three illustrations are provided as examples of how direct and indirect addresses are assigned in the global, local, and sublocal areas.

GLOBAL VARIABLES

Global data variables are assigned starting at 'G'[0] in the same order as written in the source program. Note that in the case of indirect arrays, address pointers are assigned in the direct locations; the space allocated to the arrays proper follows all of the direct (simple) variables and address pointers.

```
INT  a,          ! simple variable
```

is the first variable declared in the program and is assigned to 'G'[0].

```
     b[0:5],     ! direct array
```

is assigned to 'G'[1:6].

```
    .c[0:5],     ! indirect array
```

the address pointer ".c" is assigned to 'G'[7], the array proper is assigned following the direct and pointer variables to 'G'[10:15].

```
    .d,          ! pointer variable
```

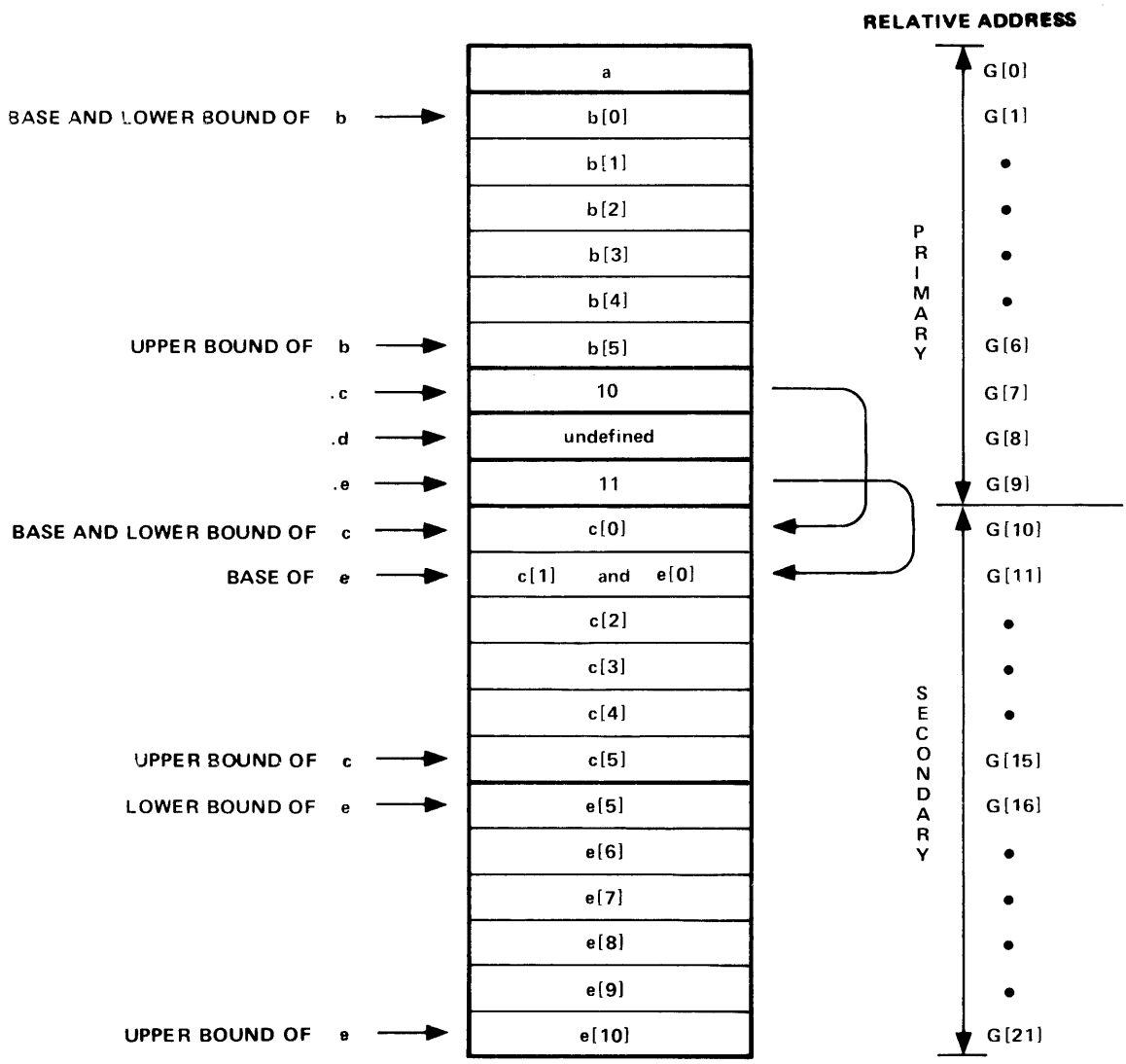
is assigned to 'G'[8].

```
    .e[5:10];    ! indirect array
```

the address pointer ".e" is assigned to 'G'[9], the array proper is assigned, following the array "c", to 'G'[16:21].

EXAMPLE OF GLOBAL ADDRESS ASSIGNMENTS

```
INT  a,          ! simple variable
     b[0:5],     ! direct array
    .c[0:5],     ! indirect array
    .d,          ! pointer variable
    .e[5:10];    ! indirect array
```



LOCAL VARIABLES

Local data variables are assigned starting at 'L'[1] in the same order as written in the source program.

```
INT  a,          ! simple variable
```

is the first variable declared in the procedure and is assigned to 'L'[1].

```
     b[0:5],     ! direct array
```

is assigned to 'L'[2:7].

```
     .c[0:5],    ! indirect array
```

the address pointer ".c" is assigned to 'L'[8]. The 'G'[0] address of the array proper is determined when the procedure executes.

```
     .d,         ! pointer variable
```

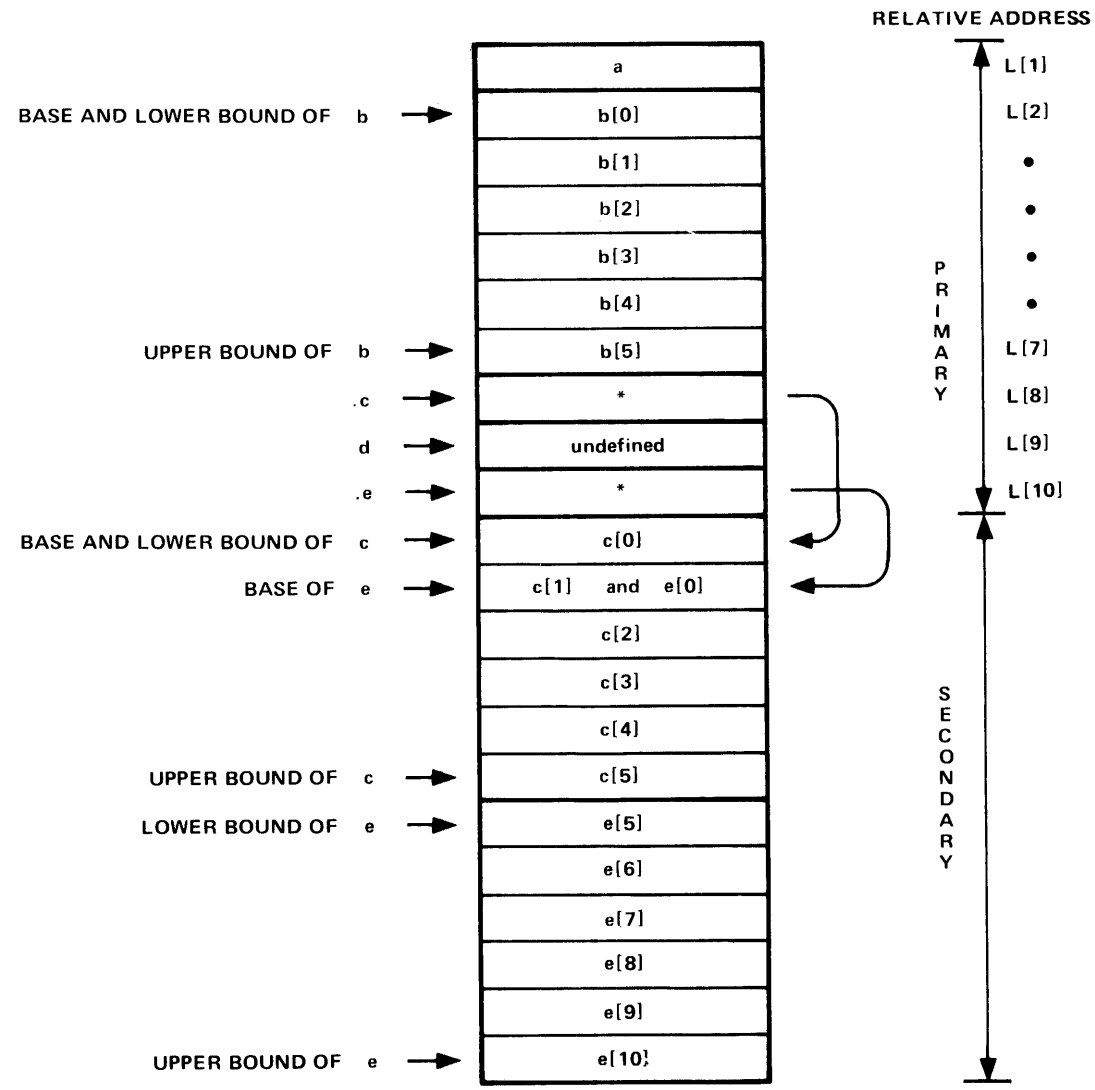
is assigned to 'L'[9].

```
     .e[5:10];   ! indirect array.
```

the address pointer ".e" is assigned to 'L'[9]. The 'G'[0] relative address of the array proper is determined when the procedure executes. Array "e" will follow array "c" in any case.

EXAMPLE OF LOCAL ADDRESS ASSIGNMENTS

```
INT  a,          ! simple variable
     b[0:5],     ! direct array
    .c[0:5],     ! indirect array
    .d,          ! pointer variable
    .e[5:10];    ! indirect array
```



* addresses of indirect local arrays are determined when the procedure begins executing

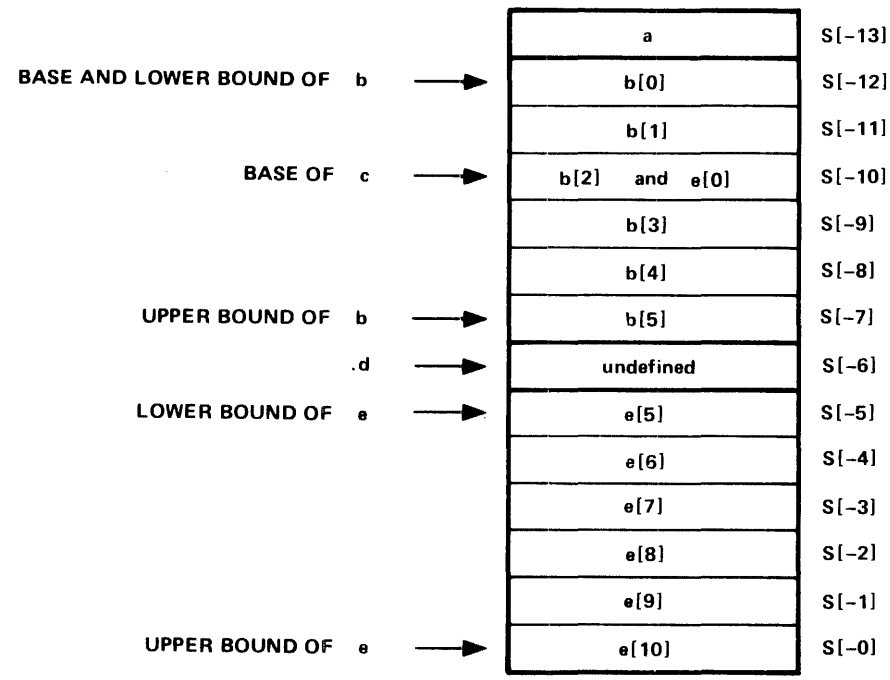
SUBLOCAL VARIABLES

In the SUBLOCAL area addresses are negative offsets from 'S'[0]. The first data variable declared is given the most negative offset from 'S'; the last variable declared is always addressed as 'S'[0]. Note that the limit of sublocal storage is 31 words.

```
   INT  a,       ! direct variable

           is assigned to 'S'[-13] because the total number of
           sublocal words of storage that are required is 14.

        b[0:5],  ! direct array

           is assigned to 'S'[-12:-7].

       .d        ! pointer variable

           is assigned to 'S'[-6].

        e[5:10]; ! direct array

           is assigned to 'S'[-5:0].
```

EXAMPLE OF SUBLOCAL ADDRESS ASSIGNMENTS

```
   INT  a,       ! direct variable
        b[0:5],  ! direct array
         .d      ! pointer variable
        e[5:10]; ! direct array
```



The LITERAL declaration assigns an integer constant value to an identifier.

The general form for a LITERAL declaration is:

```
LITERAL { <name> = <constant> } , ... ;
```

where

<name> is an identifier assigned to a LITERAL

<constant> is any value that can be represented in one word

more than one LITERAL can be specified per declaration (separated by commas ",")

example

```
LITERAL minute = 60, day = 24;
```

A literal identifier can be used anywhere an integer constant can be used. When a literal identifier is used, the assigned constant value is substituted in its place. LITERALS are only a programming convenience; they are not allocated any storage.

Note: When using a literal identifier in a byte move operation, the identifier should be surrounded by brackets "[...]" if a one-byte value is desired. If a literal value is not surrounded by brackets, it is treated as two bytes in a byte move operation.

Literals are assigned constant values:

```
LITERAL second = 1;
LITERAL minute = second * 60;
LITERAL hour = minute * 60;
```

Literals can be used in initialization:

```
INT normal^time := hour + 20 * minute;
INT maximum^time := 2 * hour;
```

Literals can be used in array bound specifications:

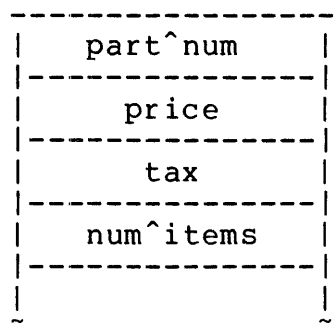
```
STRING array[0:minute];
```

is equivalent to

```
STRING array[0:60];
```

Using literals in place of constants permits significant changes to be made in a convenient manner:

Records in an inventory file are formatted as follows:



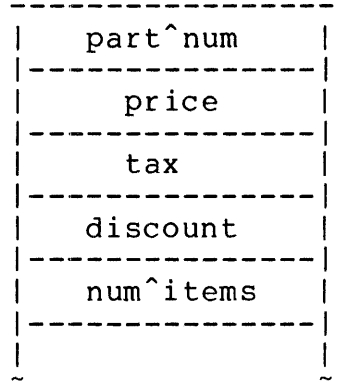
Literals could be used to define the record format to the program:

```
LITERAL part^num  = 0;
LITERAL price = part^num + 1;
LITERAL tax = price + 1;
LITERAL num^items = tax + 1;
LITERAL rec^length = num^items;

INT .inventory^array[0:num^items];
```

Then used in the program

```
inventory^array[part^num] := . . . ;
inventory^array[price] := . . . ;
inventory^array[tax] := . . . ;
inventory^array[num^items] := . . . ;
```

At some point a new item is defined and added to the inventory record format:



The literals defining the the inventory record can be changed:

```
LITERAL part^num = 0;
LITERAL price = part^num + 1;
LITERAL tax = price + 1;
   inserted => LITERAL discount = tax + 1;
    changed => LITERAL num^items = discount + 1;
LITERAL reclength = num^items;
```

The program is re-compiled to reflect the changes to the literal declarations, but other references in the program to those literals need not be changed.

A DEFINE declaration assigns a block of text to an identifier. Subsequent reference to the identifier in a context where an identifier is not being declared causes the compiler to process the block of text at the point of reference.

The general form for a DEFINE declaration is:

```
DEFINE { <name> = <block of text> # } , ... ;
------   ------ - --------------- -         -

where

   <name> is an identifier assigned to a DEFINE

   <block of text> is any character or sequence of characters
   (including literals and other defines) to be invoked when
   <name> is referenced

   more than one block of text can be defined per declaration
   (separated by number sign, comma "#,")

example

   DEFINE error = total < subtotal#, timeout = 100000D#,
          bit^field = <0:15>#;
```

No evaluation is made of the text when declared. The define is invoked where the define identifier is encountered as long as the identifier is not being declared at that point. For example

```
DEFINE error = total < subtotal#; ! global declaration.

PROC a;
  BEGIN
    INT error;

      does not invoke the define "error" because the identifier
      "error" is being declared as an INT variable in this context.

  END; ! a.

PROC b;
  BEGIN

    IF error THEN ...

       invokes the define "error" because the identifier is not
       being declared in this context.

  END; ! b.
```

When invoked, the text is processed as though it actually exists where the identifier appears (that is, the text is checked to determine if it is proper in its context and, possibly, machine instructions are emitted).

Some examples:

```
DEFINE error = total < subtotal#;
```

If used as follows

```
IF error THEN .....;
```

is equivalent to

```
IF total < subtotal THEN .....;
```

```
DEFINE timeout = 100000D#;  ! too large for LITERAL.
```

If used as follows

```
some^double := timeout;
```

is equivalent to

```
some^double := 100000D;
```

```
DEFINE bit^field = <10:15>#; ! bit extract/deposit field.
```

If used as follows

```
word.bit^field
```

is equivalent to

```
word.<10:15>
```

A define could be used to define part of a header line:

```
DEFINE version = "V0001"#;
```

If used as follows

```
STRING header[0:33] := ["TANDEM APPLICATION LANGUAGE: ",version];
```

is equivalent to

```
STRING header[0:33] := "TANDEM APPLICATION LANGUAGE: V0001";
```

Caution should be taken when using a defined identifier in a conditional expression. For example, a common operation is to

increment a variable and compare the result with a limit. This could be done using a DEFINE declaration:

```
INT a;

DEFINE inc^a = a := a + 1#;
```

IF the defined identifier "inc^a" is used by itself

```
inc^a;
```

"a" is simply incremented by one and the expected result occurs.

But if the defined identifier is used in a conditional expression, the expected result does not occur:

```
IF inc^a < 10 THEN ...
```

is equivalent to

```
IF a := a + 1 < 10 THEN ...
```

tests the result of the arithmetic expression "a + 1" for being less than 10, then assigns the result of the comparison (true, -1, or false, 0) to "a". See Conditional expressions for a more complete explanation.

The proper way to invoke this type of defined identifier in a conditional expression is to surround the identifier with parentheses. This gives the assignment operation a higher precedence than the comparison:

```
IF (inc^a) < 10 THEN ...
```

which is equivalent to

```
IF (a := a + 1) < 10 THEN ...
```

There is another form of DEFINE having parameters. The general form for a parametric define is:

```
DEFINE { <name> ( <formal parameter name> , ... )
       = <block of text> # } , ... ;
```

where

<name> is an identifier assigned to a parametric DEFINE

<formal parameter name> is a parameter. Each parameter must be referenced within <block of text>

<block of text> is any character or sequence of characters (including literals and other defines) to be invoked when <name> is referenced

example

```
DEFINE bad^result(a,b) = IF a < b THEN#;
```

Some examples are:

```
DEFINE bad^result(a,b) = IF a < b THEN#;
```

if invoked as

```
bad^result(total,subtotal) .... ;
```

is equivalent to

```
IF total < subtotal THEN .... ;
```

A commonly used operation such as initializing an array with blanks can be defined:

```
DEFINE blank^buf(array,num^blanks) =
         BEGIN
           array := " ";
           array[1] ':=' array FOR num^blanks - 1;
         END#;
```

puts a blank in [0]. Then moves [0] to [1], [1] to [2], [2] to [3], etc.

Then invoked as

```
  .
STRING in^buffer[0:71];              ! data declaration.
  .
  .
blank^buf(in^buffer,72);
```

is equivalent to

```
BEGIN
  in^buffer := " ";
  in^buffer[1] ':=' inbuffer FOR 71;
END;
```

which fills "in^buffer" with 72 blanks

A comma "," can be part of the text in the parameter part of the define if surrounded by apostrophes "'":

```
DEFINE varproc ( procname , params ) = CALL procname ( params )#;
```

then

```
varproc ( FILEINFO , fnum','error );
```

is equivalent to

```
CALL FILEINFO ( fnum , error );
```

Additionally, parentheses may be used in a parameter providing that opening and closing parens are matched.

Procedures comprise the executable part of a T/TAL program. All programs must contain at least one procedure (designated "MAIN") and typically contain many procedures.

An important characteristic of procedures is that they can be written without regard for the actual variables to be processed and that the same procedure can be used to process information involving many different sets of variables.

Other characteristics of procedures are:

* The calling environment is saved when a procedure is called and restored when a procedure finishes.

* Function procedures can be written that produce a value. The procedure name can be used like a variable in an expression.

* Variables, constants, expressions, and other procedures can be passed as parameters.

* A procedure's local variables are known only to the procedure and occupy space only while the procedure executes.

* A procedure's initialized local variables are initialized each time the procedure is entered.

* All items that can be declared globally (except procedures) can be declared locally (i.e., within a procedure).

* Procedures themselves can have subprocedures.

* Because the calling environment is saved when a procedure is called, procedures can be written that call themselves (recursive).

* A procedure's instruction codes can be made to reside in main memory at all times.

The general form of a procedure declaration is

```
+-------------------------------------------------------------------+
|                                                                   |
|  procedure heading:    gives the procedure a name and lists and   |
|                        describes any parameters                   |
|                                                                   |
|  procedure body:       contains local declarations (optional),    |
|                        subprocedure declarations (optional), and  |
|                        statements                                 |
|                                                                   |
|  procedure heading:                                               |
|                                                                   |
|    [ <type> ] PROC <name> [ <attributes> ] ;                      |
|               ---- -----                    -                     |
|  or                                                               |
|                                                                   |
|    [ <type> ] PROC <name> ( <formal parameter name> , ... )       |
|               ---- ------ - ----------------------       -        |
|       [ <attributes> ] ;                                          |
|                        -                                          |
|       <parameter specifications>                                  |
|       --------------------------                                  |
|                                                                   |
|  procedure body:                                                  |
|                                                                   |
|      BEGIN                                                        |
|      -----                                                        |
|                                                                   |
|        [ local declaration ]                                      |
|                  .                                                |
|                  .                                                |
|                  .                                                |
|        [ local declaration ]                                      |
|                                                                   |
|        [ subprocedure declaration ]                               |
|                  .                                                |
|                  .                                                |
|                  .                                                |
|        [ subprocedure declaration ]                               |
|                                                                   |
|        [ [ <statement> ] ; ]                                      |
|                  .                                                |
|                  .                                                |
|                  .                                                |
|        [ [ <statement> ] ; ]                                      |
|                                                                   |
|      END ;                                                        |
|      --- -                                                        |
|  or                                                               |
|      FORWARD ;  or  EXTERNAL ;                                    |
|      ------- -      -------- -                                    |
|                                                              -->  |
+-------------------------------------------------------------------+
```

```
example

   INT PROC find^last (array, limit) RESIDENT; ! heading.
     INT limit;                                !
     STRING .array;                            !

     BEGIN                                     ! body.
       INT addr;                               !
       RSCAN array[ limit ] WHILE " " -> addr; !
       RETURN addr '-' @array;                 !
     END;                                      !
```

The procedure heading assigns an identifier to the procedure, lists and describes any formal parameters, optionally assigns the procedure a type, and specifies any attributes.

There are two forms of a procedure heading: one for procedures without parameters; one for procedures having parameters. The forms are:

without parameters

```
[ <type> ] PROC <name> [ <attributes> ] ;
```

with parameters

```
[ <type> ] PROC <name> ( <formal parameter name> , ... )
    [ <attributes> ] ;

    <parameter specifications>
```

where

<type>, if included, means that the procedure is a function. It is one of

```
{ INT                    }
{ INT(32)                }
{ STRING                 }
{ FIXED [ ( <fpoint> ) ] }
```

<name> is the identifier assigned to the procedure

<attributes> are MAIN [, RESIDENT]

where

MAIN indicates that the procedure is the first one to execute when the program is run

RESIDENT indicates that the procedure's instruction codes are to be made main memory resident when the program is run

<formal parameter name> is the identifier that is used within the procedure body to reference the parameter. The formal parameter has the value of the actual parameter when the procedure is invoked

-->

<parameter specifications> describe each <formal parameter> by <type> and whether it is a "value" or a "reference" parameter. <parameter specifications> is of the form

```
<param type> { [ . ] <formal parameter name> } , ... ;
-------------  ----------------------          -
      .                    .                    ;
      .                    .                    ;
```

where

<parm type> is

```
{ INT                     }
{ INT(32)                 }
{ STRING                  }
{ FIXED [ ( <fpoint> ) ]  }
{       [ (     *    ) ]  }
{ PROC                    } (by value only)
{ <type> PROC             } (by value only)
```

. is the indirection symbol.

If absent, a VALUE parameter is indicated. The parameter is evaluated and the value that it represents is passed to the procedure in the parameter area (statements within the procedure body access a copy of the actual parameter)

If present, a REFERENCE parameter is indicated. The 'G'[0] relative address of the variable is put in the parameter area (and statements within the procedure body access the variable indirectly through the parameter location). If the parameter is type STRING, then a byte address is passed. Otherwise, a word address is passed

examples

```
PROC aproc;

INT PROC find^last (array, limit) ! function proc with params.
   MAIN, RESIDENT;                ! attributes.
                                  ! parameter specifications.
   INT limit;                     !   value.
   STRING .array;                 !   reference.
```

<type>

If a procedure is assigned a type, it is a function procedure and using its name in an expression causes it to be invoked. A value of the specified type (i.e., INT, INT(32), STRING, or FIXED) is returned and used in the expression where the procedure name exists. The value is returned to the expression through use of a RETURN statement in the body of the function procedure.

In the preceding example, the type is INT and the value returned is a result of the expression

```
addr '-' @array                    ! "addr" minus "@array".
```

<name>

<name> is an identifier used to name the procedure. <name> is used when invoking the procedure with a CALL statement or, if the procedure is a function, when it is referenced in an expression.

```
INT length := 71;                  ! data declarations.
STRING .buffer  [ 0:72 ];          !

INT PROC find^last (array, limit) RESIDENT; ! heading.
  INT limit;                               !
  STRING .array;                           !

  BEGIN                                    ! body.
    INT addr;                              !
    RSCAN array[ limit ] WHILE " " -> addr; !
    RETURN addr '-' @array;                !
  END;                                     !
```

could be called by using

```
CALL find^last ( buffer, length);
```

however, the value assigned to "find^last" is lost.

or in an expression

```
IF (find^last( buffer, length ) ) > 0 THEN .....;
```

<attributes>

The <attributes> specify certain operating environments of a procedure. The <attributes> are:

RESIDENT When the program is run, the procedure's instruction codes will be made to reside in main memory at all times (will not be written over). Resident procedures will be placed

as the first (physically) procedures in a program (just after global read-only arrays).

MAIN — This indicates that the procedure is to be the first one executed when the program is run. More than one procedure can be designated a "MAIN" procedure. However, the last procedure designated "MAIN" is the the one that is first executed when the program is run.

Additionally, when any procedure designated "MAIN" is finished (i.e., last statement is executed or a RETURN statement is encountered) a call is made to the Guardian Operating System STOP procedure if the STOP procedure has been declared as an external procedure. This stops program execution and causes a message to be sent to the creating process.

The procedure body contains statements that are executed when the procedure is called. The body can contain data, define, literal, entry, and/or subprocedure declarations as well as any T/TAL statements.

Statements in a procedure are executed until either the last statement is executed or a RETURN statement is encountered. Program execution then returns following the point where the procedure was invoked (unless the procedure was designated MAIN; in that case, program execution stops).

The general form of the procedure body is:

```
      BEGIN
      -----

        [ <local declaration> ]
                  .
                  .
                  .
        [ <local declaration> ]

        [ subprocedure declaration ]
                  .
                  .
                  .
        [ subprocedure declaration ]

        [ [ <statement> ] ; ]
                  .
                  .
                  .
        [ [ <statement> ] ; ]

      END ;
      --- -
  or
      FORWARD ;  or  EXTERNAL ;
      ------- -      -------- -

  where

     <local declaration> is data, LITERAL, DEFINE, or ENTRY

     FORWARD indicates that the actual procedure declaration
     is located later in the source program

     EXTERNAL indicates that the actual procedure is part of the
     operating system
                                                              -->
```

```
example

    BEGIN
      RETURN a^char - b^char;
    END;
```

Note: The storage available for directly addressed local variables is limited to 127 words. Indirectly addressed local variables occupy space in the secondary local area (which is limited to the first 32,768 words in the data area).

Any item declared locally in the procedure body is known only to the procedure and subprocedures declared within the procedure. If an item is declared locally that has the same name as a global item, the locally declared item is used within the procedure body (and any associated subprocedures).

FORWARD AND EXTERNAL

The purpose of declaring a procedure with the word EXTERNAL or FORWARD in place of the body, is to inform the compiler that a particular identifier is a procedure name and specify what parameters are required to call that procedure.

EXTERNAL is used if the actual procedure is part of the operating system. Procedures callable by application programs that are part of the operating system are sharable by all programs running on the system.

```
PROC READ (filenum, buf, count, countread, tag) CALLABLE, VARIABLE;
   INT filenum, .buf, count, .countread;
   INT(32) tag;
   EXTERNAL;
```

then called somewhere in the program

```
   .
CALL READ(in^file, in^buffer, 72, num^read);
   .
```

Note: So that the external declarations for the callable operating system procedures won't have to be entered into each source program, there is a Tandem-supplied file containing all of the external declarations for the callable procedures.

This file is designated -

$SYSTEM.SYSTEM.EXTDECS

and can be invoked using the compiler SOURCE command

```
?SOURCE $SYSTEM.SYSTEM.EXTDECS
```

which compiles the external declarations for all of the operation system callable procedures into the application program

Or a particular external declaration can be selected using the SOURCE command and specifying the operating system procedure name:

```
?SOURCE $SYSTEM.SYSTEM.EXTDECS(READ)
```

compiles the external declaration for the READ procedure into the application program.

FORWARD is used if the actual procedure declaration (having the actual body) is to be found further down in the source program. This is necessary so that a procedure can call another procedure that has not yet been actually declared. This is desirable for a number of reasons:

* so that two procedures can call each other

* So that procedures can be ordered in a program in a manner that enhances clarity or efficiency

An example:

```
PROC anyproc; FORWARD;
```

Another example:

```
INT PROC first ( p1, p2 );
   INT p1, p2;
   FORWARD;

PROC second;

   BEGIN

     INT local^int;                  ! local data declaration.
       .
     local^int := first(3,6);        ! call to "first" procedure.
       .
   END;

INT PROC first ( p1, p2);
   INT p1, p2;

   BEGIN                             ! actual body of "first".
       .
       .
     RETURN p1 + p2;
   END;
```

Each procedure parameter must be assigned a <formal parameter name> and be described in the <parameter specification>.

<formal parameter names>

The formal parameter names provide local identifiers so that parameters can be referenced within the procedure body. A formal parameter assumes the value of the actual (i.e., calling) parameter when the procedure is invoked. The formal parameter names have meaning only within the body of a procedure, so can duplicate global identifiers.

In the preceding example, the formal parameters are "array" and "limit". The actual variables are "buffer" and "length".

<parameter specifications>

This part of the procedure heading describes each parameter (as to its type) and indicates whether a parameter is to be passed to the procedure by value or by reference:

```
<param type> { [ . ] <formal parameter name> } , ... ;
-------------          -----------------------          -
      .                         .                       ;
      .                         .                       ;
```

<parm type>

Two classes of <parm type> are permitted - data variables and procedures:

```
PROC my^proc (int^param, int32^param, string^param, fixed^param,
              proc^param, s^proc^param);
   INT int^param;                    ! integer parameter.
   INT(32) int32^param;              ! double integer parameter.
   STRING string^param;              ! string parameter.
   FIXED(3) fixed^param;             ! fixed parameter.
   PROC proc^param;                  ! procedure parameter.
   STRING PROC s^proc^param;         ! type proc parameter.
```

An identifier assigned to a <formal parameter> is treated as the specified <parm type> within the body of the procedure.

For example:

```
PROC p (a,b,c);
  INT a;
  PROC b;
  INT PROC c;
```

"a" is specified as <parm type> INT, "b" is specified as <parm type> PROC, "c" is specified as <parm type> INT PROC (function procedure).

When used in the body of the procedure

```
BEGIN
  INT n;

    n := a;

       "a" is treated as a variable with a data type of INT.  The
       value that a represents is stored into the local variable
       "n".

    CALL b;

       "b" is treated as a procedure with the name of "b".  The
       actual procedure that "b" represents is invoked.

    n := c;

       "c" is treated as an type INT function procedure.  The
       actual procedure that "c" represents is invoked.  The
       value that it returns is stored into "n".

END; ! of "p"
```

Notes:

1. Multiple element variables (i.e., arrays) must be passed by reference if more than a single element is to be accessed.

2. Procedures are passed by value. Any parameters of a passed procedure are treated as type INT by value.

3. Read-only array variables cannot be passed as parameters.

4. The <fpoint> of a FIXED variable passed as reference parameter must match the <fpoint> value in the parameter specification. If they do not match NO scaling is performed and the compiler issues a warning message; statements within the procedure body treat the variable as though is has the <fpoint> specified in the parameter specification.

5. FIXED operands passed as value parameters are scaled up or down as necessary to match the <fpoint> specified in the parameter specification. If the <fpoint> value of an operand passed as a parameter is greater than that of the <fpoint> specified in the parameter specification, the internal representation of the operand will be scaled down and precision will be lost.

6. A special form of <parm type> for FIXED parameters is permitted. It is FIXED(*). This form permits a parameter having any <fpoint> value to be passed without having the compiler emitting instructions to scale the operand. Within

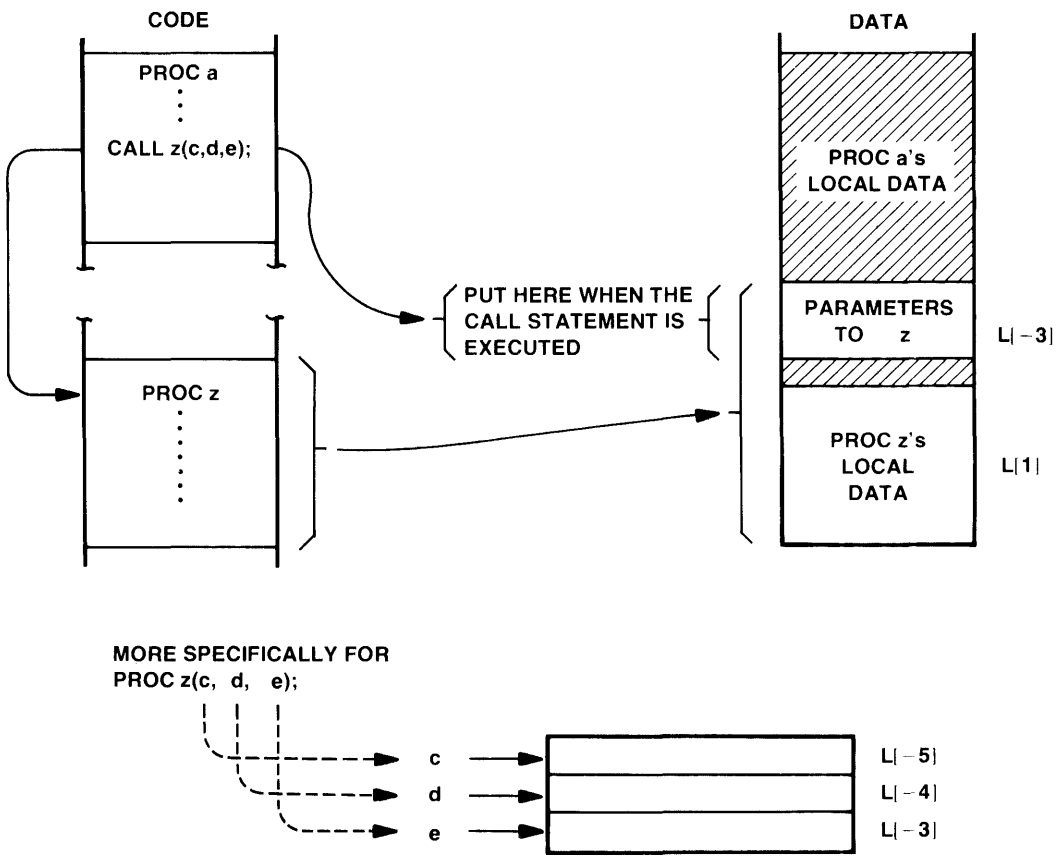
the procedure body, the parameter will be treated as though is has an <fpoint> of zero.

7. INT, INT(32), and FIXED variables can be passed for STRING reference parameters. The compiler will emit code to convert the word address to a byte address.

Parameter Area

Parameters are passed to a procedure in the data area preceding its local data area. Twenty-nine words are available in this area for parameter passing. INT and STRING value parameters use one word each, INT(32) value parameters use two words each, FIXED value parameters use four words each. All reference parameters use one word each.

PARAMETER AREA



INT, INT(32), STRING, and FIXED Value Parameters

If the indirection symbol is absent, a VALUE parameter is indicated. The parameter is treated as an expression and the value that it represents is passed to the procedure in the parameter area (and statements within the procedure body access the parameter value location directly).

Note: A value parameter can be used as working space within the body of the procedure without affecting the actual variable(s) used to generate the value for that parameter.

Examples of value parameters:

```
INT varyl := 1, vary2;                ! data declaration.

PROC some^proc (first^one, second^one);
   INT first^one;                     ! value parameter.
   INT .second^one;                   ! reference parameter.

  BEGIN
    .
    .
  END;
```

When invoking a procedure, the names of variables can be used for value parameters:

```
CALL some^proc(varyl, vary2);
```

The value represented by "varyl" is assigned to "first^one".

An expression can be used as a value parameter:

```
CALL some^proc( 2 * varyl, vary2);
```

the value assigned "first^one" is 2 times "varyl".

Another form of expression used as a value parameter:

```
CALL some^proc(IF vary2 > 0 THEN 1 ELSE 2 , varyl);
```

the value assigned "first^one" is conditional, dependent on the value of "vary2".

A function procedure or subprocedure can be used as a value parameter:

```
CALL some^proc(find^last,vary2);
```

"find^last" is called and executed, its value is then assigned to "first^one".

Note: An anomaly exists when passing and referencing STRING value parameters. The compiler converts the STRING parameter to an integer value (i.e., <param>.<0:7> = 0, <param>.<8:15> = parameter). When the parameter is referenced in the procedure body, it is again treated as a STRING variable (i.e., <param>.<0:7>). This results in the value of the STRING parameter being lost. Therefore, if STRING value parameters are used, the STRING value should be passed in the form

<expression> '<<' 8

which shifts the significant portion of the STRING value into <param>.<0:7>.

INT, INT(32), STRING, and FIXED Reference Parameters

If the indirection symbol is present, a REFERENCE parameter is indicated. The 'G'[0] relative address of the variable is put in the parameter area (and statements within the procedure body access the variable indirectly through the parameter location). Values can be stored in the actual variables represented by reference parameters.

An example:

```
INT array [0:9];                        ! data declaration.

PROC some^proc(first^one, second^one);
   INT first^one,
       .second^one;

  BEGIN
     .
    second^one[5] := 0;
     .
  END;
```

then "some^proc" is invoked as follows:

```
CALL some^proc(varyl, array[3]);
```

puts the value 0 into "array[8]".

A constant, representing a 'G'[0] relative address, can be passed as a reference parameter (the compiler will issue a warning message):

```
   .
CALL some^proc(varyl, 10);
   .
```

puts the value 0 into 'G'[15]

A function procedure or subprocedure that returns a 'G'[0] relative address can be passed as a reference parameter (the compiler will issue a warning message):

For example

```
STRING .array[0:72] := "        VERSION 1 ", ! global declaration
       .out^buffer[0:72];                    ! global declaration

INT PROC find^start (buffer);
   STRING .buffer;

  BEGIN

    INT start;

    SCAN buffer WHILE " " -> start;
    RETURN start;
  END;
```

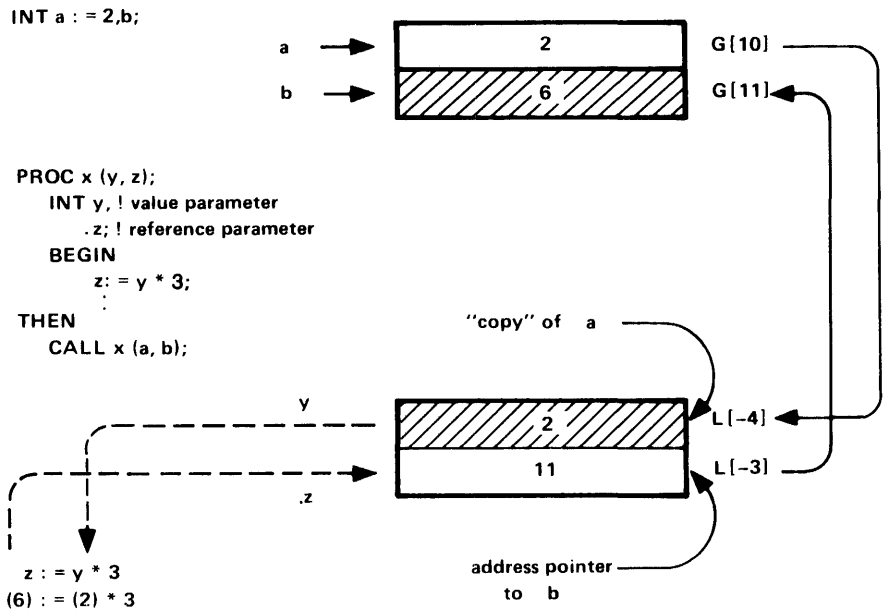
Another procedure is declared as follows:

```
PROC format^line(format);
   STRING .format; ! reference parameter.

  BEGIN

    out^buffer ':=' "HEADER " & format FOR 10;

       moves the string "HEADER" into "out^buffer", followed by ten
       bytes starting from the address represented by "format".

  END;
```

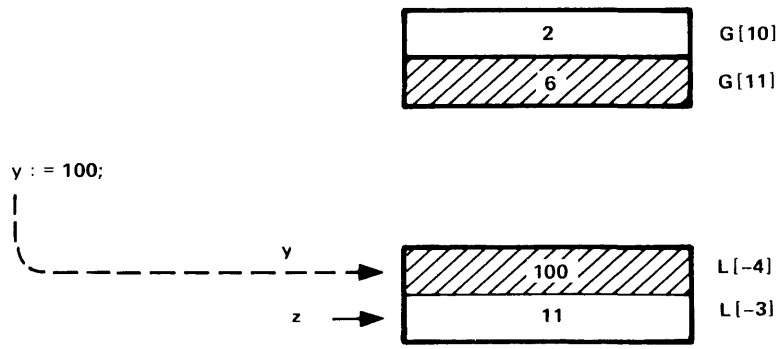
Then "format^line" is invoked as follows:

```
CALL format^line(find^start(array));

   which results in the global array "out^buffer" containing

"HEADER VERSION 1 "
```

VALUE VERSUS REFERENCE PARAMETERS



USING A VALUE PARAMETER AS LOCAL STORAGE



<type> PROC Value Parameters

A procedure can be specified as a <formal parameter>. The identifier associated with a <type> PROC <formal parameter> is treated as a procedure name within the procedure body.

For example:

```
PROC z(pname);
   PROC pname;

  BEGIN
     .
    CALL pname;

        invokes the procedure represented by "pname"
  END;
```

The actual procedure passed is invoked at the time of the call

```
 PROC a;  ! procedure declaration
   BEGIN
     .
   END;

 PROC b;  ! procedure declaration
   BEGIN
     .
   END;
```

Then

```
CALL z(a);

   "z" executing is equivalent to

      BEGIN
         .
        CALL a;

             procedure "a" is invoked
```

OR

```
CALL z(b);

   "z" executing is equivalent to

      BEGIN
         .
        CALL b;

             procedure "b" is invoked
```

If the actual procedure(s) to be passed, themselves have parameters, the programmer must be certain that all parameters are supplied (the compiler cannot check). These parameters are treated as type INT value parameters.

For example:

```
PROC c (g,h,i)
   INT g,h,i;
  BEGIN
    .
  END;
```

In this case, a procedure that would be passed procedure "c" as a PROC parameter, must pass three parameters to the procedure it invokes:

```
PROC y(pname);
   PROC pname;

  BEGIN
    INT p1,p2,p3;

    CALL pname(p1,p2,p3);

       The parameters - "p1", "p2", and "p3" - are passed to the
       procedure represented by "pname" as type INT value
       parameters.
```

If the actual procedure to be invoked has reference parameters, the parameters must be preceded by an at "@" symbol in the call:

```
PROC d (j,k,l);
   INT .j, .k, .l; ! reference parameters
  BEGIN

    .
  END;
```

In this case, a procedure that calls "d" must be written as follows:

```
PROC x(pname);
   PROC pname;

  BEGIN
    INT p1,p2,p3;

    CALL pname(@p1,@p2,@p3);

       passes the 'G'[0] relative addresses of the parameters -
       "p1", "p2", and "p3" - to the procedure represented by
       "pname".

  END;
```

Functionally, a subprocedure is quite similiar to a procedure. That is, a subprocedure is a contiguous block of machine instructions called (with parameters) to perform a specific operation.

Some characteristics of subprocedures are:

* Subprocedures are invoked and exited faster than procedures.

* Like procedures, the calling environment is saved when the subprocedure is called and restored when the subprocedure finishes (permitting recursive subprocedures).

* Function subprocedures can be written that return a value to the subprocedure name when used in an expression.

* Variables, constants, expressions, and procedures can be passed as parameters.

* Sublocal variables are known only to the subprocedure and occupy space only while the subprocedure is active.

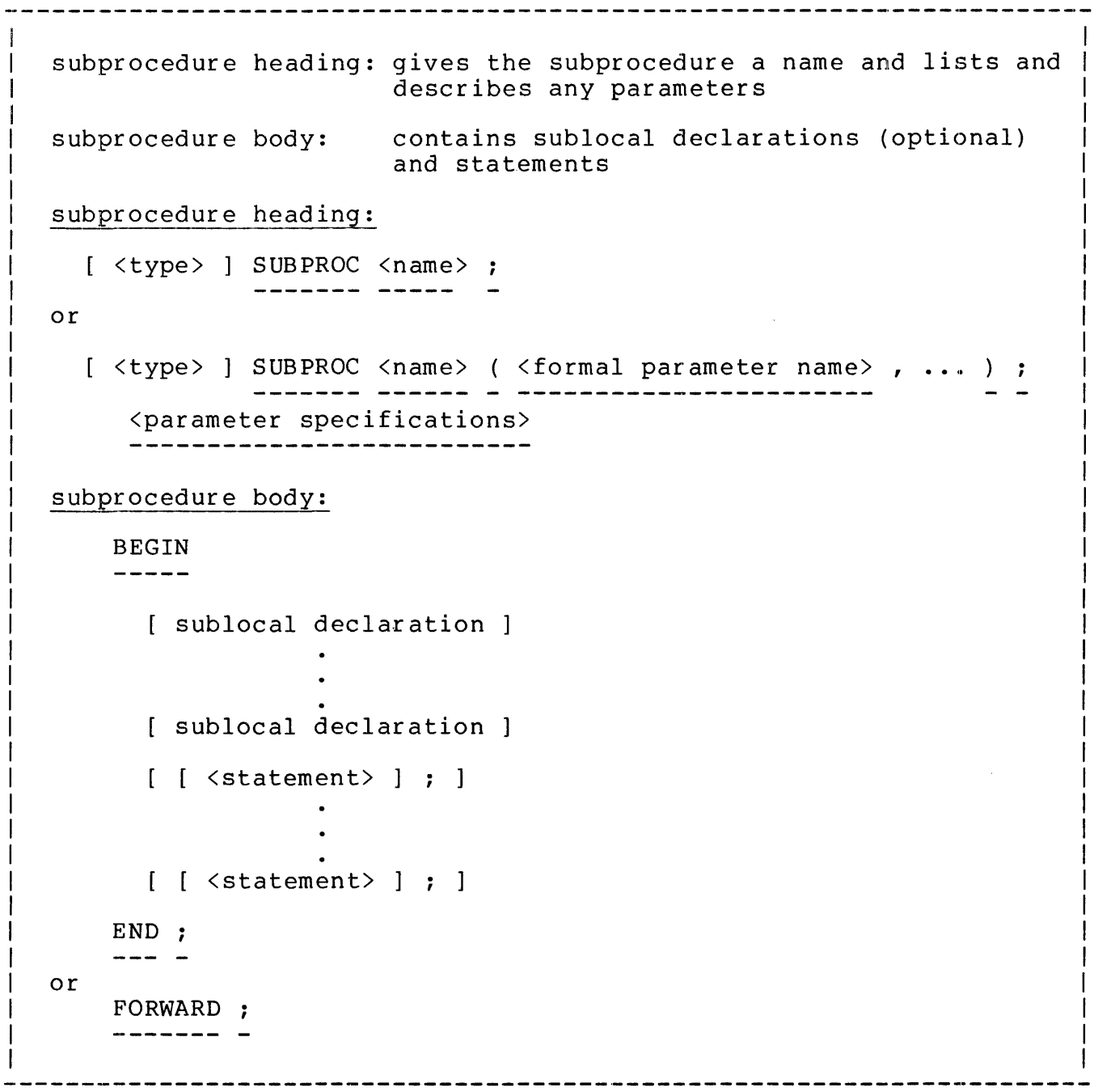
* A subprocedure's initialized local variables are initialized each time the subprocedure is entered.

* All items that can be declared globally (except procedures) can be declared sublocally (i.e., within a subprocedure).

Because subprocedures can directly access the procedure's local variables and are entered and exited faster than procedures, they are typically used when a specific operation is needed at various points within a particular procedure body. (If such an operation is needed throughout a program, a procedure must be used.)

SUBPROCEDURE DECLARATION

The general form of a subprocedure declaration is



The subprocedure heading is like a procedure heading: it assigns an identifier to the subprocedure, lists and describes any formal parameters, and optionally assigns the subprocedure a type.

There are two forms of a subprocedure heading: one for subprocedures without parameters; one for subprocedures having parameters.

The forms are:

```
    ! without parameters.

  [ <type> ] SUBPROC <name> ;

    ! with parameters.

  [ <type> ] SUBPROC <name> ( <formal parameter name> , ... ) ;
     <parameter specifications>

  where

     <type>, if included, means that the subprocedure is function.
     It is one of

        { INT                    }
        { INT(32)                }
        { STRING                 }
        { FIXED [ ( <fpoint> ) ] }

     <name> is the identifier assigned to the subprocedure

     <formal parameter name> is the identifier that is used within
     the subprocedure body to reference the parameter.  The formal
     parameter has the value of the actual parameter when the
     subprocedure is invoked

     <parameter specifications> describe each <formal parameter> by
     <type> and whether it is a "value" or a "reference" parameter.
     <parameter specifications> is of the form

        <param type> { [ . ] <formal parameter name> } , ... ;
              .                      .                 ;
              .                      .                 ;

        where
```

-->

```
<parm type> is

   { INT                       }
   { INT(32)                   }
   { STRING                    }
   { FIXED [ ( <fpoint> ) ]    }
   {       [ (     *    ) ]    }
   { PROC                      } (by value only)
   { <type> PROC               } (by value only)

.  is the indirection symbol.

   If absent, a VALUE parameter is indicated.  The
   parameter is evaluated and the value that it
   represents is passed to the subprocedure in the
   parameter area (statements within the subprocedure
   body access a copy of the actual parameter)

   If present, a REFERENCE parameter is indicated.  The
   'G'[0] relative address of the variable is put in the
   parameter area (and statements within the
   subprocedure body access the variable indirectly
   through the parameter location).  If the parameter is
   type STRING, then a byte address is passed.
   Otherwise, a word address is passed

examples

   SUBPROC aproc;

   INT SUBPROC find^last (array, limit) ! function with params.
      MAIN, RESIDENT;                   ! attributes.
                                        ! parameter specifications.
      INT limit;                        !   value.
      STRING .array;                    !   reference.
```

<type>

The functions and characteristics of a subprocedure assigned a type are identical to that of a function procedure. That is, if a subprocedure is assigned a type, it is a function subprocedure and its <name> can be used in an expression. A value of the specified type (i.e., INT, INT(32), or STRING) is returned in place of the subprocedure <name> through use of a RETURN statement.

<name>

The functions and characteristics of the subprocedure <name> are identical to the procedure <name>. That is, <name> is an identifier used to name the subprocedure. <name> is used when invoking the subprocedure with a CALL statement or, if the subprocedure is a function, when it is referenced in an expression.

<formal parameters>

The functions and characteristics of a subprocedure's formal parameters are similiar to a procedure's. They provide sublocal identifiers so that the subprocedure can be written without regard for actual variable names. The formal parameters are then used in expressions in the subprocedure body (global and local variables can also be used if desired). The formal parameters assume the value of the actual (i.e., calling) parameters when the subprocedure is invoked. The formal parameters have meaning only within the body of a subprocedure, therefore they can duplicate global and local identifiers.

<parameter specifications>

The functions and characteristics of <parameter specifications> are identical to the corresponding part in the <procedure heading>. That is, this part of the subprocedure head describes each parameter (as to its <parm type>) and indicates whether a parameter is to be passed to the subprocedure by value or by reference.

Notes:

1. Multiple element variables (i.e., arrays) must be passed by reference if more than element [0] is to be accessed.

2. Procedure names must be passed by value. Any parameters of a passed procedure are treated as type INT by value.

3. A value parameter can be used as working space within the body of a subprocedure without affecting the actual variable(s) used to generate the value for that parameter.

4. A parameter must be declared by reference if a value is to be stored into the actual variable that the parameter represents.

5. Read-only array variables cannot be passed as parameters.

6. The <fpoint> of a FIXED variable passed as reference parameter must match the <fpoint> value in the parameter specification. If they do not match NO scaling is performed and the compiler issues a warning message; statements within the subprocedure body treat the variable as though is has the <fpoint> specified in the parameter specification.

7. FIXED operands passed as value parameters are scaled up or down as necessary to match the <fpoint> specified in the parameter specification. If the <fpoint> value of an operand passed as a parameter is greater than that of the <fpoint> specified in the parameter specification, the internal representation of the operand will be scaled down and precision will be lost.

8. A special form of <parm type> for FIXED parameters is permitted. It is FIXED(*). This form permits a parameter having any <fpoint> value to be passed without having the compiler emitting instructions to scale the operand. Within the subprocedure body, the parameter will be treated as though is has an <fpoint> of zero.

9. INT, INT(32), and FIXED variables can be passed for STRING reference parameters. The compiler will emit code to convert the word address to a byte address.

10. If STRING value parameters are used, the STRING value should be passed in the form

    <expression> '<<' 8

The subprocedure body is like a procedure body; it contains statements that are executed when the subprocedure is called. The body can contain data, define, literal, and entry declarations as well as any T/TAL statements.

Statements in a subprocedure are executed until either the last statement is executed or a RETURN statement is encountered. Program execution then returns following the point where the subprocedure was invoked.

There is also a special form of the body to tell the compiler that the actual subprocedure body will be found further along in the compilation (FORWARD).

The general form of the subprocedure body is

```
      BEGIN
      -----

        [ <sublocal declaration> ]
                    .
                    .
                    .
        [ <sublocal declaration> ]

        [ [ <statement> ] ; ]
                    .
                    .
                    .
        [ [ <statement> ] ; ]

      END ;
      --- -
  or
      FORWARD ;
      ------- -

  where

     <sublocal declaration> is data, LITERAL, DEFINE, or ENTRY

     FORWARD indicates that the actual subprocedure declaration
     is located later in the source program

  example

     BEGIN                                          ! body.
       array := " ";                                !
       array [ 1 ] ':=' array FOR length - 1; !
     END;                                           !
```

Note: The total storage available for a subprocedure's sublocal variables and parameters is limited to 31 words. Indirect arrays cannot be declared in a subprocedure.

Any item declared sublocally in the subprocedure body is known only to the subprocedure. If an item is declared sublocally that has the same name as a global item, the sublocally declared item is used within the subprocedure body.

Like a procedure, the purpose of declaring a subprocedure with the word FORWARD in place of the body, is to inform the compiler that a particular identifier is a subprocedure name and specify what parameters are required to call that subprocedure. This is necessary so that a subprocedure can call another subprocedure that has not yet been actually declared (and so two subprocedures can call each other).

An ENTRY declaration is used to specify additional entry points (i.e., start of execution) into a procedure or subprocedure body. When an ENTRY declaration is used to call a procedure or subprocedure, it is treated as though it is the actual name (i.e., the formal parameters and specifications apply to the <entry point name>).

The general form of an ENTRY declaration is:

```
ENTRY <entry point name> , ... ;
----- ------------------       -

where

   <entry point name> is a label in the procedure or
   subprocedure body that indicates an entry point and an
   identifier used when invoking the procedure or subprocedure

   more than one entry point can be specified per declaration
   (separated by commas ",")

example

   ENTRY first^entry, second^entry;
```

For example, a subprocedure could be written with multiple entry points:

```
SUBPROC blanks ( array, length );
   STRING .array;
   INT length;

   BEGIN

     ENTRY points, dashes;

     array := " ";
     GOTO spread;

   points:

     array := ".";
     GOTO spread;

   dashes:

     array := "-";

   spread: array[ 1 ] ':=' array FOR length - 1;
   END;
```

then could be called in the following forms:

```
   CALL blanks ( buffer , num^chars ); ! execution begins with
                                       ! first statement.
```

or

```
   CALL points ( inarray , limit );    ! execution begins with
                                       ! "points:".
```

or

```
   CALL dashes ( outarray , 71 );      ! execution begins with
                                       ! "dashes:".
```

If an entry point to a procedure (or subprocedure) body is to be called prior to where the actual body exists, the entry point must be declared FORWARD. This is done by substituting the entry point name for the procedure or subprocedure name in a FORWARD declaration.

Using the example entry point "points", the following FORWARD declaration would be made:

```
   SUBPROC points( array, length );
      STRING .array;
      INT length;

     FORWARD;
```

T/TAL provides the following bit-level operations:

* Bit Extraction (one word quantities only)

* Bit Deposit (one word quantities only)

* Bit Shift (one or two word quantities)

Using bit extraction and bit deposit permits operations on portions of words rather than just an entire word.

Bit extraction permits access to a portion of a word. The result of the extraction is treated as though it is right justified with the unspecified left bits set to 0.

The general form for bit extraction is:

```
<primary> . "<" <left bit> [ : <right bit> ] ">"
--------- -  -  ----------                   -
```

where

<primary> is a constant, variable, function procedure, <arithmetic expression> in parentheses, <assignment statement> in parentheses, standard function, or a bit function. (The <primary> is not altered by bit extraction.)

<left bit> is an integer constant specifying the left bit of the bit field

<right bit> is an integer constant specifying the right bit of the bit field

example

```
some^vary.<8:11>                ! accesses bits 8 through 11.
```

Note:

Because of the way the hardware handles STRING variables, the left bit of a STRING variable should be considered bit 8 when using bit extraction.

example

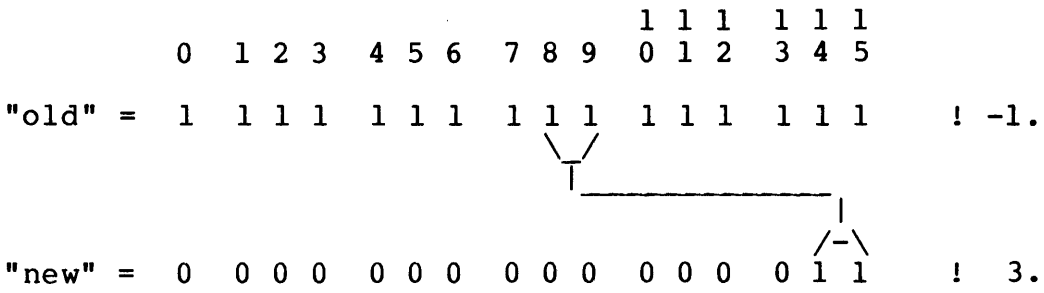
```
STRING str^vary;                ! data declaration.

IF str^vary.<8> = 0 THEN ..;    ! checks leftmost bit of
                                ! <str^vary>.
```

```
INT old := -1, new := 0;        ! data declaration.
```

then extracting bits 8 and 9 and assigning the result

```
new := old.<8:9>;
```

results in



bit extraction can be used in a conditional expression:

```
IF some^word.<0:7> = "A" THEN ...;
```

checks bits 0 through 7 for the ASCII character "A".

or

```
IF some^word.<15> THEN... ;
```

checks bit 15 of "some^word" for non-zero value.

Bit extraction can be used on array elements:

```
vary := some^array[8].<8:15>;
```

accesses right half of "some^array[8]".

Bit extraction can be used on expressions:

```
result := (first^num + second^num).<4:7>;
```

"first^num" and "second^num" are added together, bits 4 through 7 of the result are assigned to "result".

or

```
IF (result := (first^num + second^num).<4:7>) > 0 THEN ...;
```

this is the same as the previous example except that the value assigned to "result" is also checked for being greater than 0.

Bit Deposit

Bit deposit provides a method to assign a value to a specific portion of a word. Values are assigned through use of an assignment statement.

The general form for bit deposit is:

```
<variable> . "<" <left bit> [ : <right bit> ] ">" := <expression>
```

where

<left bit> is an integer constant specifying the left bit of the bit field

<right bit> is an integer constant specifying the right bit of the bit field

:= is the assignment operator

example

```
some^vary.<8:11> := %17;          ! sets bits 8 thru 11 to 1's
```

Note

Because of the way the hardware handles STRING variables, the left bit of a STRING variable should be considered bit 8 when using bit deposit.

example

```
STRING str^vary;                  ! data declaration.
str^vary.<8> := 0;                ! sets leftmost bit of
                                  ! <str^vary> to 0.
```

```
INT old := -1;
```

is represented as

```
                                  1 1 1  1 1 1
          0  1 2 3  4 5 6  7 8 9  0 1 2  3 4 5

"old" =   1  1 1 1  1 1 1  1 1 1  1 1 1  1 1 1    ! -1.
```

then assigning the value 0 to bits 10 and 11 to "old"

```
old.<10:11> := 0;
```

results in

```
"old" =   1  1 1 1  1 1 1  1 1 1  0 0 1  1 1 1    !  %177717.
```

Bits, within one or two word elements, can be shifted a specified number of positions to the left or right by applying a shift operator.

The general form for using shift operators is:

```
<primary> <shift op> <number of positions>
--------- ---------- --------------------

where

   <primary> is a constant, variable, function procedure,
   <arithmetic expression> in parentheses, <assignment
   statement> in parentheses, standard function, or a bit
   function.  (The <primary> is not altered by bit shift.)

   <shift op> is

      <<    signed left shift (sign bit not changed, 0's filled
            from right)

     '<<'   unsigned left shift (through bit 0, 0's filled
            from right)

      >>    signed right shift (sign bit propagated)

     '>>'   unsigned right shift (from bit 0, 0's filled
            from left)

   <number of positions> is a <primary> indicating the number
   of positions to be shifted

example

   new := old >> 3;                  ! arithmetic right shift,
                                     ! three positions.
```

The type of the <primary> determines whether the shift operates on one or two words.

An example of signed left shift versus unsigned left shift:

```
   INT old := %33333, new;           ! data declaration.
```

is represented as

```
                                    1 1 1  1 1 1
          0  1 2 3  4 5 6  7 8 9  0 1 2  3 4 5

   "old" =  0  0 1 1  0 1 1  0 1 1  0 1 1  0 1 1    ! %33333.
```

then performing a signed left shift two positions

```
new := old << 2;
```

results in

```
"new" =  0  1 0 1  1 0 1  1 0 1  1 0 1  1 0 0    ! %55554.
```

or performing an unsigned left shift two positions

```
new := old '<<' 2;
```

results in

```
"new" =  1  1 0 1  1 0 1  1 0 1  1 0 1  1 0 0    ! %155554.
```

An example of signed right shift versus unsigned right shift:

```
INT old := -256, new;                ! data declaration.
```

is represented as

```
                                    1 1 1  1 1 1
          0  1 2 3  4 5 6  7 8 9  0 1 2  3 4 5
"old" =  1  1 1 1  1 1 1  1 0 0  0 0 0  0 0 0    ! %177400.
```

then performing a signed right shift five positions

```
new := old >> 5;
```

results in

```
"new" =  1  1 1 1  1 1 1  1 1 1  1 1 1  0 0 0    ! %177770.
```

or performing an unsigned right shift five positions

```
new := old '>>' 5;
```

results in

```
"new" =  0  0 0 0  0 1 1  1 1 1  1 1 1  0 0 0    ! %3770.
```

Bit shifts can be used on any <primary> including partial word quantities:

```
INT first, second := -1;             ! data declaration.
```

then extracting bits 8 through 15 and performing an signed left shift three positions

```
first := second.<8:15> << 3;
```

results in

```
"first" =  0  0 0 0  0 1 1  1 1 1  1 1 1  0 0 0    ! %3770.
```

<number of positions> can be any <primary> such as an expression.

```
first := second.<8:15> '>>' (2 * some^vary);
```

the number of positions shifted is the value of 2 times "some^vary".

or a function procedure

```
INT PROC num^bits;
  BEGIN
    .
  RETURN varyl + vary2;
  END;

first := second.<8:15> << num^bits;
```

the number of positions shifted is determined by the value returned from "num^bits".

An expression is a sequence of operations upon constants and variables that can be evaluated to a single value or state. Expressions are used in certain program components (those components, in turn, can be used in expressions). Two types of expressions are used in T/TAL; arithmetic expressions (evaluates to a value) and conditional expressions (evaluates to a state):

```
<expression> is

   <arithmetic expression>

   <conditional expression>
```

Arithmetic expressions are used to compute values that are assigned to variables, to index array elements, to assign values to value parameters when invoking procedures and subprocedures, and to supply values for certain parts of statements.

A typical use of an arithmetic expression is in an assignment statement:

```
INT result, variable;                ! data declaration.

result := 3 * variable / 2;

   the result of the expression (3 times "variable" divided by 2)
   is stored in "result".
```

Conditional expressions are used in statements to make decisions concerning the path of program execution.

A typical use of a conditional expression is in an IF statement:

```
IF result > variable THEN ...;

   the conditional expression ("result > variable") is
   evaluated.  If the expression is true the statement following
   THEN is executed.
```

Three hardware indicators are subject to change as the result of the instructions executed to evaluate an arithmetic or conditional expression. They are:

* Condition Code Indicator (CC) -- generally, indicates if the result of an operation was a negative value, zero, or a positive value. When a value is assigned to a variable through use of an assignment statement, the condition code reflects the new value in the variable. The condition code indicator can be checked by using one of the relational operators "< = > <= >= <>" in a conditional expression.

* Carry -- indicates that a carry out of the high order bit position occurred. The carry indicator can be checked by using the standard function "$CARRY" in a conditional expression.

* Overflow -- indicates that the result of an operation could not be represented in the available number of bits in the data format. The overflow indication can be checked using the standard function "$OVERFLOW" in a conditional expression.

  An Overflow condition causes an interrupt to the operating system Overflow trap handler.

If a hardware indicator is to be checked, it must be done before another arithmetic operation is performed.

An arithmetic expression is a rule for computing a single numeric value of a specific data type. More than one type of data is not permitted in an expression unless a type transfer function is used. There are four (interchangeable) forms of arithmetic expression:

* General form
* Assignment form
* IF THEN form
* CASE form

The general form of an arithmetic expression is:

```
[ + | - ] <primary> [ { <arith op> <primary> } ... ]
          ---------
where

   + - are unary plus and minus indicating the sign of the
   leftmost <primary>.  If omitted, plus is assumed

   <primary> is one or more syntactic elements that represents a
   single value.  <primary> has several forms, they are:

      <constant>

      <variable>

      function procedure

      standard function

      bit function

      ( <arithmetic expression> )

   <arith op> - arithmetic operator - is:

      {  +   }  signed add
      {  -   }  signed subtract
      {  *   }  signed multiply
      {  /   }  signed divide
      {      }
      { '+'  }  unsigned add
      { '-'  }  unsigned subtract
      { '*'  }  unsigned multiply
      { '/'  }  unsigned divide
      { '\'  }  unsigned modulo divide (provides remainder)
      {      }
      { LOR  }  logical or
      { LAND }  logical and
      { XOR  }  exclusive or
                                                              -->
```

examples

```
   varyl                ! <primary> only form.
   - varyl              ! - <primary> form.
   - varyl * 2          ! +- <primary> <arith op> <primary> form.
   varyl + vary2        ! <primary> <arith op> <primary> form.
   varyl * (-vary2)     ! <primary> <arith op> <primary> form - the
                        ! right <primary> is an arithmetic
                        ! expression in parentheses.
```

PRIMARY

A <primary> is one or more syntactic elements that represent a single value. Some examples of <primarys> are

<primary>	example
<constant>	10 ! decimal integer constant.
<variable>	vary[10] ! variable with index.
function procedure	find^last (buffer, length)
standard function	$MIN (a,b)
bit function	vary[10].<8:15>
(<arithmetic expression>)	(vary[10] + 10) ! in parentheses.

An <arithmetic expression> in parentheses is treated as a single entity. This is sometimes used when an arithmetic expression contains of more then one <primary>. For example

```
2 * (3 + 4)
```

The <primary> indicated by "(3 + 4)" is evaluated. The result is then multiplied by "2". (If the parentheses were omitted, the multiplication would be performed first, then the result of the multiplication would be added to "4".)

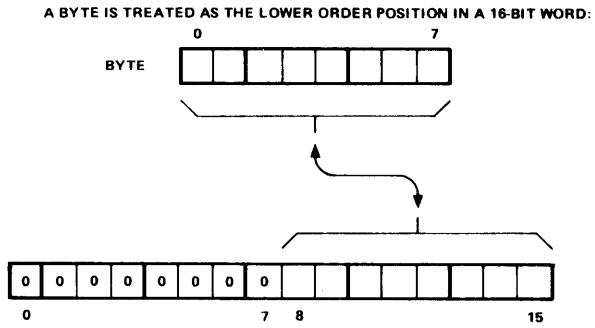
ARITHMETIC OPERATORS

The arithmetic operators (i.e., add, subtract, multiply, divide, and the logical operators) are combined with <primarys> to form arithmetic expressions. Three kinds of arithmetic operations are provided: signed arithmetic on INT, INT(32), and FIXED quantities; unsigned arithmetic on INT quantities; and logical operations on INT quantities. Note that signed arithmetic, unsigned arithmetic, and

logical operations can be mixed in an expression (provided that the proper data types are used).

Note: Byte operands are treated as the least significant half (i.e., word.<8:15>) of an INT operand (the most significant half is set to zero). Normal INT signed or unsigned arithmetic or logical operations are performed as indicated.

BYTE OPERANDS



Signed Arithmetic

Signed arithmetic is indicated by using the arithmetic operators not surrounded by apostrophes "'" (legal operand types are indicated):

<arith op>		legal operations
+	signed add:	INT + INT, INT(32) + INT(32), and FIXED + FIXED
-	signed subtract:	INT - INT, INT(32) - INT(32), and FIXED - FIXED
*	signed multiply:	INT * INT and FIXED * FIXED
/	signed divide:	INT / INT and FIXED / FIXED

If the result of a signed operation can not be represented within the number of bits indicated by the operand type (i.e., 15 bits for an INT, 31 bits for an INT(32), 63 bits for a FIXED), an Overflow occurs. Overflow also occurs if a divide operation is attempted with divisor of 0 (zero).

Some examples of signed arithmetic:

```
INT integer1, integer2;              ! data declarations.
INT(32) double1, double2;            !
STRING byte1, byte2;                 !

integer1 * integer2 + integer1;
integer2 / integer1;
double1 + double2;

byte1 + byte2
integer1 / byte1
```

STRING variables treated as right half of INT variables.

these are invalid:

integer1 + double1

requires type transfer.

double1 / double2

INT(32) divide not permitted.

double1 * double1

INT(32) multiply not permitted.

Unsigned Arithmetic

Unsigned arithmetic is indicated by using the arithmetic operators surrounded by apostrophes "'" (legal operand types are indicated):

<arith op>		legal operations
'+'	unsigned add:	INT '+' INT only
'-'	unsigned subtract:	INT '-' INT only
'*'	unsigned multiply:	INT '*' INT only (INT(32) result)
'/'	unsigned divide:	INT(32) '/' INT only (INT result)
'\'	unsigned modulo divide: (returns remainder)	INT(32) '\' INT only (INT result)

The results obtained from a unsigned add and subtract are identical to that obtained by signed add and subtract except that unsigned add and subtract do not set the Overflow indicator. The 16-bit result, the condition code setting, and the Carry indicator setting are the same. Unsigned divide ('/' or '\'), however, sets the Overflow indicator if the quotient cannot be represented in 16 bits.

Typically, unsigned arithmetic is used when operating with quantities whose values ranges from 0 to 65,535. This is particularly useful when dealing with pointer variables because they contain 16-bit addresses.

Some examples of unsigned arithmetic (using the preceding data declarations):

```
integer1 '+' integer2
integer1 '-' byte1
integer1 '*' integer2
```

provides INT(32) product.

```
double1 '\' integer1
```

provides INT remainder.

these are invalid:

integer1 '\' integer2

requires type transfer.

double1 '+' double2

unsigned INT(32) operations not permitted (except left side of modulo divide).

integer1 '*' integer2 '+' integer1

unsigned multiply provides an INT(32) result and type mixing is not permitted (type transfer could be used)

Logical Operations

The logical operations (i.e., LOR, LAND, and XOR) apply to INT quantities only.

An example of LOR:

```
STRING ascii^char := "A";           ! data declaration.

ascii^char LOR %40
```

logically "or" %40 to value of "ascii^char".

is equivalent to

```
"ascii^char"    0 1  0 0 0  0 0 1   ! big "A".
LOR %40         0 0  1 0 0  0 0 0

results in      0 1  1 0 0  0 0 1   ! little "a".
```

An example of LAND:

```
STRING ascii^char := "a";           ! data declaration.

ascii^char LAND %337
```

logically "and" %337 to value of "ascii^char".

is equivalent to

```
"ascii^char"    0 1  1 0 0  0 0 1   ! little "a".
LAND %337       1 1  0 1 1  1 1 1

results in      0 1  0 0 0  0 0 1   ! big "A".
```

An example of XOR:

```
INT word1 := %052525, word2 := %031463;
                                        ! data declaration.
word1 XOR word2
```

is equivalent to

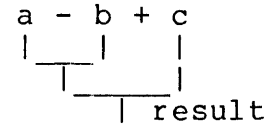
```
"word1"      0  1 0 1  0 1 0  1 0 1  0 1 0  1 0 1  ! %052525.
XOR "word2"  0  0 1 1  0 0 1  1 0 0  1 1 0  0 1 1  ! %031463.

results in   0  1 1 0  0 1 1  0 0 1  1 0 0  1 1 0  ! %063146.
```

PRECEDENCE OF OPERATORS

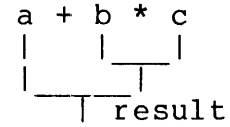
To avoid ambiguity, arithmetic operations and bit functions are performed according to a precedence. An operation can be given precedence over other operations by enclosing it in parentheses. The precedence is:

1. Bit extraction or deposit ! highest precedence.
2. Bit shift
3. multiply and divide operators (* / '*' '/' '\')
4. add and subtract operators (+ - '+' '-') and logical operations (LOR LAND XOR)

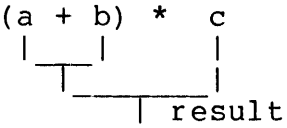
Operations having equal precedence are performed from left to right:



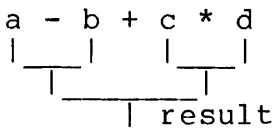
Operations having highest precedence are performed first:



Operations can be enclosed in parentheses, giving them a higher precedence than operations outside of parentheses:



Left to right operation occurs until a higher precedence operator is encountered:



HOW STRING ELEMENTS ARE TREATED IN EXPRESSIONS

When a string element is used in an arithmetic expression, it is treated as the right-half of an INT variable with the left half set to zeros:

For example, a string variable is stored in memory as follows:

```
STRING byte := "W";                    ! data declaration.
```

is represented as

```
                                       1 1 1  1 1 1
            0  1 2 3  4 5 6  7 8 9  0 1 2  3 4 5

"byte" =    0  1 0 1  0 1 1  1 0 0  0 0 0  0 0 0  ! "W" (%127).
```

then when used in an arithmetic expression, "byte" is treated as

```
            0  0 0 0  0 0 0  0 0 1  0 1 0  1 1 1  ! %000127.
```

HOW FIXED OPERANDS ARE SCALED IN EXPRESSIONS

FIXED variables and constants used in FIXED calculations need not have the same <fpoint> value. Instead, the compiler emits instructions to scale the operands when addition or subtraction is performed or, when multiplication or division is performed, determines a new <fpoint> value for the result.

For addition or subtraction:

So that precision is not lost, the operand having smaller <fpoint> value is scaled up to match the larger <fpoint> value. The <fpoint> value of the result of an addition or subtraction is the same as the larger <fpoint> value. For example, performing the addition

```
3.005F + 6.01F
```

causes the right operand to be scaled up by a factor of one. So that, in effect, the following calculation is actually made:

```
3.005F + 6.010F
```

and the result is 9.015F. Keep in mind, however, that the result would be scaled down if assigned to a variable with an <fpoint> less than 3.

For multiplication:

- The operands are not scaled when a multiplication is performed. However, the <fpoint> value of the result of a multiplication is the sum of the <fpoint> values of the two operands. For example, performing the multiplication

```
3.091F * 2.56F     ! FIXED(3) * FIXED(2)
```

  results in the FIXED(5) value 7.91296F.

For division:

- The operands are not scaled when a division is performed. However, the <fpoint> value of the result is the difference of the <fpoint> value of the dividend minus the <fpoint> value of the divisor. For example, performing the division

```
4.05F / 2.10F      ! FIXED(2) / FIXED(2)
```

  results in the FIXED(0) value 1. Note that precision is lost. Remember that integer division is actually being performed and the internal representation of those two operands is actually

```
405 / 210
```

  To retain precision when performing division with operands having nonzero <fpoint> values, the <fpoint> value of the dividend should be scaled up by a factor equal to the <fpoint> value of the divisor. This can be accomplished by using the standard function $SCALE. For example, using the same values as in the preceding example:

```
$SCALE(4.05F,2) / 2.10F
```

  scales the dividend up by a factor of two. This is equivalent to the expression

```
4.0500F / 2.10F
```

  the result of which is the FIXED(2) value 1.92F.

The following example illustrates the scaling involved in evaluating an expression containing FIXED operands having different <fpoint> values. The example also shows how the result of an expression is scaled to match the variable it is assigned to.

```
FIXED     a;          ! data declarations.
FIXED(2)  b;          !
FIXED(-1) c;          !

a := 2.015F * (b + c);
        |       |   |
        |       | up 3   <c> is scaled up by three to match <b>,
        |       |___|    which is 2
        |         |
      3 |_________| 2    the result of the multiplication is an
            | 5          implied <fpoint> of 5
            |
          down 5         the result of the expression is scaled
a <---------|            down by 5 to match <a>.  Note that
                         precision is lost
```

HOW FUNCTION PROCEDURES ARE USED IN EXPRESSIONS

If a function procedure or subprocedure is used in an expression, it is called and executed in the process of evaluating the expression:

```
LITERAL records^per^block = 16;     ! data declarations.
INT com^account = 123;              !
INT item^number;                    !

INT PROC block^no ( acct^no );
   INT acct^no;
   BEGIN
     RETURN acct^no * 2;
   END;
```

Then "block^no" is used in an expression and evaluated as shown:

```
block^no ( com^account ) * records^per^block + item^number
          |                |                  |
       BEGIN               |                  |
         RETURN 123 * 2;   |                  |
       END;                |                  |
          | result is 246  | 16               |
          |________________|                  |
                   | result is 3936           | 11
                   |__________________________|
                              | result is 3947
```

Arithmetic Expressions: assignment form

The assignment form of arithmetic expression permits an assignment statement to be used as an arithmetic expression.

Its form is:

```
<variable> [ . "<" <left bit> ">" [ : "<" <right bit> ">" ] ]
----------
   := <arithmetic expression>
   -- ----------------------

where

   <variable> is a declared data variable with an optional bit
   deposit field

   :=  is the assignment operator

   <arithmetic expression> represents a value of the same type as
   <variable>

   The resultant value of this form of arithmetic expression is
   the value of <variable> after the assignment

example

   a := a + 1
```

This type of arithmetic expression is used when the value assigned to a variable can also be used where an expression is required.

For example, the <index> used to indicate an offset from a variable can be an arithmetic expression. A method to automatically reference the next element in an array is

```
IF array[a := a + 1] <> 0 THEN ...
```

"a := a + 1" is an assignment statement that is used as an arithmetic expression. Each time the IF statement is executed, the next element of "array" is referenced.

The IF .. THEN form of arithmetic expression permits a conditional expression to select either of two arithmetic expressions.

Its form is:

```
IF <conditional expression> THEN <arithmetic expression>
                            ELSE <arithmetic expression>

where

  <conditional expression> is evaluated to determine the
  <arithmetic expression> to be computed.

example

    IF length > 0 THEN 10 ELSE 20
```

If <conditional expression> is evaluated as being true, then the <arithmetic expression> following THEN is computed. If false, the <arithmetic expression> following ELSE is computed.

This type of arithmetic expression is typically used to select either of two values to be assigned through use of an assignment statement.

For example

```
a := IF num = 10 THEN 20 ELSE 30;
```

If the variable "num" has the value of 10 then the value 20 is assigned to "a", otherwise the value 30 is assigned.

```
a := IF flag THEN b ELSE c;
```

If the variable "flag" contains a nonzero value (true), the value contained in the variable "b" is assigned, otherwise the value in "c" is assigned.

If surrounded by parentheses, this form of <arithmetic expression> can be mixed with the other form:

```
vary * index + (IF index > limit THEN vary * 2 ELSE vary * 3)
```

The CASE form of arithmetic expression permits the value of one arithmetic expression to select one expression from a list of arithmetic expressions.

Its form is:

```
CASE <index> OF
BEGIN
  <expression for index = 0> ;
  <expression for index = 1> ;
     .
     .
     .
  <expression for index = n> ;
  [ OTHERWISE <expression> ; ]
END
```

where

<index> is an arithmetic expression indicating which of the list of arithmetic expressions is to be evaluated

OTHERWISE (which is optional) indicates an alternate arithmetic expression to be evaluated if <index> does not point to one of the expressions

example

```
CASE a + 2 OF
  BEGIN
    b;
    c;
    d;
    OTHERWISE 0;
  END
```

A typical use of the case form of arithmetic expression would be to select one of a number of values to be assigned through use of an assignment statement.

For example:

```
i := CASE a OF
       BEGIN
         b;
         c;
         d;
         OTHERWISE -1;
       END;
```

If the value of the variable "a" is 0, the value of "b" is assigned.

If the value of "a" is 1, the value of "c" is assigned.

If the value of "a" is 2, the value of "d" is assigned.

If "a" has any other value, the value of -1 is assigned.

A conditional expression can evaluate the relationship between two expressions or arrays, test an expression for a non-zero state, and/or check hardware condition code settings. The result of a conditional expression is a true (indicating that the tested condition was found) or false state.

The form of a conditional expression is:

```
[ NOT ] <condition> [ { { AND | OR } [ NOT ] <condition> } ... ]
```

where

<condition> is one or more syntactic elements that represents a single state. <condition> has four forms, they are:

- <relation> ! tests hardware condition code.
- <arithmetic expression>
- <arithmetic expression> <relation> <arithmetic expression>
- (<conditional expression>)
- array comparison ! see "Array Comparison".

where

<relation> is

{ < }	signed less than
{ = }	signed equal to
{ > }	signed greater than
{ <= }	signed less than or equal
{ >= }	signed greater than or equal
{ <> }	signed not equal
{ }	
{ '<' }	unsigned less than
{ '=' }	unsigned equal
{ '>' }	unsigned greater than
{ '<=' }	unsigned less than or equal
{ '>=' }	unsigned greater than or equal
{ '<>' }	unsigned not equal

NOT means that <condition> is tested for a false state

AND means both <conditions> must be true

OR means either <condition> can be true

-->

examples

```
a                 ! <condition> only form.
NOT a             ! NOT <condition> only form.
a OR b            ! <condition> OR <condition> form.
a AND b           ! <condition> AND <condition> form.
a AND NOT b OR c  ! <condition> AND [NOT] <condition> ... form.
```

HOW CONDITIONAL EXPRESSIONS ARE EVALUATED

Conditional expressions are evaluated from left to right (according to precedence of associated operators). <conditions> combined using OR operators are evaluated until a condition is found true; other associated "ored" <conditions> are not evaluated. <conditions> combined using AND operators are evaluated until a condition is found false; other associated "anded" <conditions> are not evaluated.

A numerical value is associated with the true and false states resulting from the evaluation of a conditional expression. If a conditional expression evaluates to a true state, a value of -1 (%177777) is produced; a conditional expression evaluating to a false state produces a value of zero. The value can be used in an arithmetic expression.

For example:

```
a := b OR c;
```

The conditional expression is "b OR c." If either "b" or "c" is a nonzero value (true), the conditional expression evaluates to a true state and a -1 is assigned to "a". If both "b" and "c" are zero (false), the conditional expression evaluates to a false state and a zero is assigned to "a".

CONDITIONS

A <condition> is one or more syntactic elements that represents a single state. Some examples of <condition> are:

<condition>	example
<relation>	< ! tests condition code.
<arithmetic expression>	a + b
<arith exp> <relation> <arith exp>	a + b <> c + d
(<conditional expression>)	(a + b <> c + d)
array comparison	array1 = array2 FOR 10

A <condition> consisting of only a <relation> is used to test the hardware condition code indicator. For example, a common operation after calling a file management procedure is to test the condition code for a less-than state:

```
CALL READ(fnum,buffer,count);
IF < THEN ...
```

The <condition>, indicated by the <relation> "<" is true if the hardware condition code is set to CCL.

A <condition> consisting of a single <arithmetic expression> is used to test the expression for a true state. For example,

```
INT a;
  .
  .
IF a THEN ...
```

The <condition> indicated by the <arithmetic expression> "a" is true if the variable "a" contains a non-zero value.

A <condition> consisting of <arith exp> <relation> <arith exp> is used to compare to the relative values of two expressions. For example,

```
INT a,b;
  .
  .
IF a = 10 THEN
```

The <condition> indicated by "a = 10" is true if the value of "a" is decimal 10.

```
IF a <> b THEN ..
```

The <condition> indicated by "a <> b" is true if the value of "a" is not the same as the value of "b".

A <conditional expression> in parentheses is treated as a single entity. This is sometimes used when a conditional expression contains of more then one <condition>. For example

```
INT a,b;

IF NOT (b OR c) THEN ..
```

The <condition> indicated by "(b OR c)" is evaluated. Then the "NOT" operator is applied. Therefore, the expression "NOT (b OR c)" is true if both "b" and "c" are false. (This is equivalent to "NOT b AND NOT c").

A <condition> that is an array comparison is used to compare the contents of two arrays. Array comparison is described in this section under the heading "Comparing Arrays".

CONDITIONAL OPERATORS

The conditional operators (i.e., <relation>, NOT, AND, and OR) are combined with <conditions> to form conditional expressions. The <relation> operators permit two expressions to be compared in either of two ways: signed comparison on 16- or 32-bit expressions (i.e., INT and INT(32)) and unsigned comparison on 16-bit expressions (i.e., INT only).

<relation> operators not surrounded by apostrophes "'" compare signed quantities; those surrounded by apostrophes compare unsigned quantities.

Note that type mixing is not permitted within conditional expressions, but type transfer functions can be used.

Some examples:

```
INT neg := -255, pos := 256;        ! data declaration.
```

is equivalent to

```
                                    1 1 1  1 1 1
          0  1 2 3  4 5 6  7 8 9   0 1 2  3 4 5

"neg"     1  1 1 1  1 1 1  1 0 0   0 0 0  0 0 1   ! %177401.

"pos"     0  0 0 0  0 0 0  1 0 0   0 0 0  0 0 0   ! %000400.
```

then performing the signed comparison

```
neg < pos
```

results in a true state. But performing the unsigned comparison on the full 16-bit quantity

```
neg '<' pos
```

results in a false state.

some^int

the condition is true if "some^int" is a non-zero value.

<

the condition is true if the hardware condition code setting is less than.

first <= middle AND middle <= last

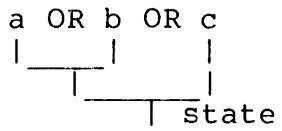
the condition is true if "middle" is in the range of "first" and "last".

PRECEDENCE OF OPERATORS

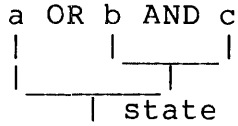
To avoid ambiguity, conditional expressions are evaluated according to a precedence. Precedence can be altered by use of parentheses. The precedence is:

1. Evaluate <condition> ! highest precedence.
   a. Evaluate <arithmetic expression>
   b. <relation> operators
2. NOT (complement) operator
3. AND (conjunction) operator
4. OR (disjunction) operator ! lowest precedence.

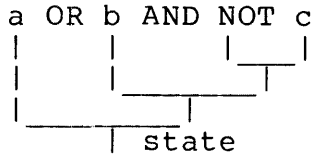
Conditions having equal precedence are evaluated from left to right:



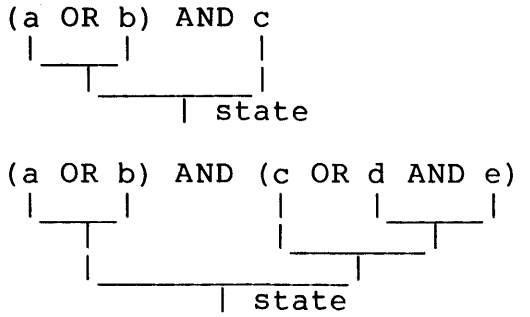
Conditions having precedence are evaluated first:



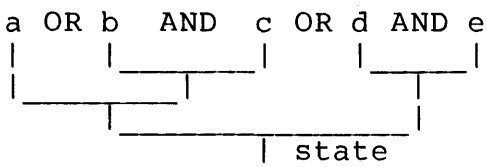
or



Conditions can be enclosed in parentheses, causing them to be evaluated before other conditions:



Conditions are evaluated from left to right until a higher precedence condition is encountered:



USING CONDITIONAL EXPRESSIONS

There are some precautions to be taken when using conditional expressions, especially in the nature of precedence of operations.

For example:

```
IF a := b = 0 THEN ..
```

evaluates the conditional expression "b = 0" and stores either a minus one or a zero into "a" depending on the result.

```
IF (a := b) = 0 THEN ..
```

stores the value of "b" into "a" then checks for equality with zero.

The same is true when mixing arithmetic expressions with conditional expressions:

```
IF a := a + 1 < 10 THEN ..
```

tests the result of the arithmetic expression "a + 1" for being less than 10 then assigns the outcome of the test (true, -1, or false, 0) to "a".

```
IF (a := a + 1) < 10 THEN ..
```

performs the arithmetic operation and assignment in parenthesis, then compares the result with 10.

Conditional expressions using the AND operator are evaluated from left-to-right until a false (zero) condition is found. It may be desireable, in some cases, to force the entire expression to be evaluated.

For example, two function procedures are used in a conditional expression:

```
IF fprocl AND fproc2 THEN ..
```

If the value returned from "fprocl" is zero, the conditional expression is false and the procedure "fproc2" is not invoked.

To force "fproc2" to be invoked, the LAND arithmetic operator could be used (and, logically, the results would be the same):

```
IF fprocl LAND fproc2 THEN ..
```

invokes both procedures.

A similar situation exists between OR and LOR.

The array comparison form of <condition> is used to compare the contents of two arrays (of any data type) with each other or to compare the contents of an array with a constant. This form of <condition> is:

```
                        { <s array> FOR <number of elements> }
  <d array> <relation>  { <constant>                         }
  --------- ----------    ---------------------------------
     [ -> <next address> ]
```

where

<number of elements> is an <arithmetic expression> of the general form specifying the maximum number of elements in <s array> to be compared. An unsigned comparison is performed

<next address> is a variable that will contain the address of the first element in <d array> that did not match the corresponding element in <s array>. If <d array> is a word-addressed variable, <next address> is a word address; if <d array> is a byte-addressed variable, <next address> is a byte address

Word-addressed arrays of different data types (e.g., INT and INT(32)) can be compared with each other. The maximum number of elements to be compared is dependent on the data type of <s array>

Byte-addressed arrays can only be compared with each other or with constant values

At the end of an array comparison, the hardware condition code indicator will be set as follows:

```
   <   (CCL) if <d array> '<' <s array>
   =   (CCE) if <d array>  =  <s array>
   >   (CCG) if <d array> '>' <s array>
```

example

```
in^array = file^name FOR 9
```

Notes:

1. During an array comparison, the elements being compared are treated as unsigned values.

2. If a read-only array is to be compared, it must be specified as the "s array".

3. When a comparison involves an INT(32) array or a FIXED array, a word comparison is actually performed. This means that the <next address> variable at the conclusion of such a comparision may not point to an element (i.e., double- or quadruple-word) boundary.

Array comparisons are typically done by using an IF THEN statement. Here are some examples:

```
IF file^name = [ "$RECEIVE" , 8 * ["  "]] THEN ..
```

uses a constant list.

```
IF in^array <> compare^mask FOR (2 * some^vary / 3) THEN ..
```

uses an arithmetic expression to determine the maximum number of element to be compared

Array comparisons can be mixed with other <conditions>:

```
IF length > 0 AND name = user^name FOR 8 AND NOT abort THEN ..

IF (file = "TERM" OR file = "term") AND mode = 5 THEN ..
```

USING <next address>

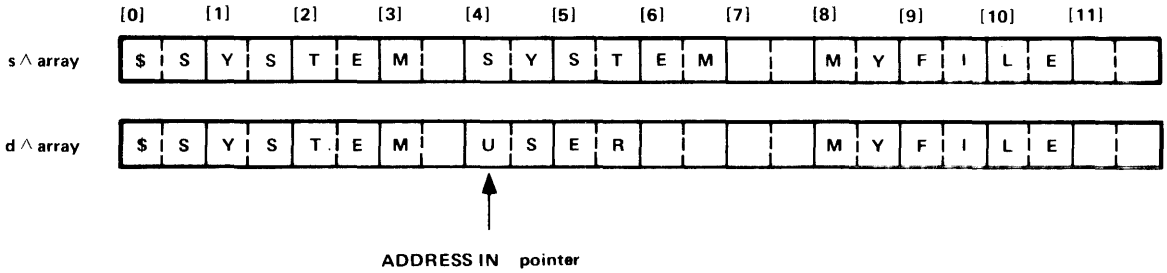
<next address> provides the 'G'[0] relative address of the first element of <d array> that did not match the corresponding element of <s array>.

EXAMPLE OF "NEXT ADDRESS" USAGE

```
INT .s^array [0:11] := "$SYSTEM  SYSTEM   MYFILE  ",
    .d^array [0:11] := "$SYSTEM  USER     MYFILE  ",
    .pointer;
```

Then performing the array comparison

```
IF d^array = s^array FOR 12 -> @pointer THEN ..
```



causes the comparison to stop with element [4]. "pointer" then contains the address of "d^array[4]".

The address in "pointer" can then be used to determine the number of elements that matched:

```
n := @pointer '-' @d^array;
```

stores the value four into the variable "n".

CHECKING CONDITION CODE

If the hardware condition code indicator is to be checked, it must be checked before an arithmetic operation is performed or a value is assigned to a variable.

For example

```
IF d^array = s^array FOR 10 -> @pointer THEN
  BEGIN ! they matched
    .
    .
  END
ELSE
  IF < THEN ...
```

If the conditional expression "<" is true, the value of the element in "d^array" that did not match is less than the corresponding element in "s^array" (using unsigned comparison).

Some characteristics of data access:

* A variable is accessed by using an identifier in a statement or expression:

```
var1 := 2;
var2 := var1;
```

  Data is stored in a variable by using an assignment statement.

* Individual array elements are accessed by appending a indexing subscript to an array identifier:

```
array[3] := 2;
var2 := array[3];
```

  An indexing subscript can be appended to a variable identifier to provide access to data elements relative to the variable:

```
var1[3] := 2;
var2 := var1[3];
```

* A direct variable can be used as an address pointer by preceding the identifier with the symbol for indirection (a period - .):

```
.var1 := 2;
var2 := .var1;
```

* The address of a variable can be obtained by preceding the identifier with the symbol for removing indirection (an at - @):

```
var2 := @var1;
```

  Data can be stored in an address pointer by preceding the identifier with the symbol for removing indirection:

```
@pointer := 2;
```

A variable is accessed by using its name in a statement or expression. The data type a variable represents is specified when declared (but a variable can be treated as another data type through the use of a type transfer function). The memory location that a variable represents can be changed through use of an assignment statement.

The general form for accessing a variable is:

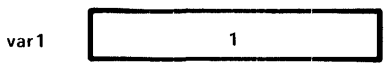
```
<variable> [ "[" <index> "]" ]
----------

where

   <variable> is simple, array, or pointer variable

   <index> is an integer constant or arithmetic expression
   specifying an element in an array whose base is <variable>.
   Normally, <index> is used only with array and pointer
   variables

example

   var := 1; array[index] := 1;
```

SIMPLE VARIABLES WITHOUT INDEX

Using a simple variable in a statement or expression references the variable.

EXAMPLE OF SIMPLE VARIABLE ACCESS WITHOUT INDEX

```
    INT var1;  ! data declaration.

 If assigned a value through an assignment statement:

    var1 := 1;
```



puts the value of 1 into the word represented by "var1"

ARRAY VARIABLES WITHOUT INDEX

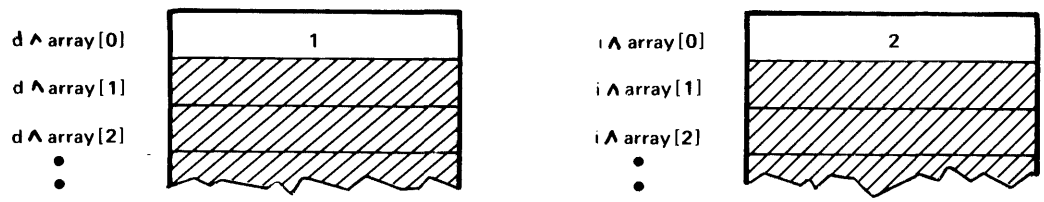
Using an array variable in a statement or expression references element [0] of the array.

EXAMPLE OF ARRAY ACCESS WITHOUT INDEX

```
INT  d^array[0:9], ! direct array.
    .i^array[0:9]; ! indirect array.
```

If assigned a value through an assignment statement:

```
d^array := 1;
i^array := 2;
```



puts the value of 1 into the word represented by "d^array[0]" and 2 into the word represented by "i^array[0]"

POINTER VARIABLES WITHOUT INDEX

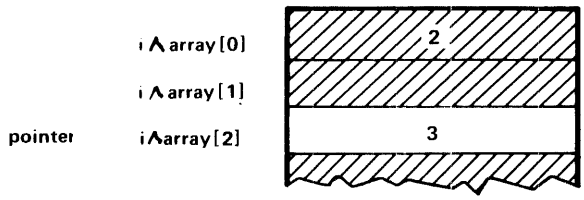
Using a pointer variable in a statement or expression accesses the variable pointed to by the address in the pointer.

EXAMPLE OF POINTER USAGE WITHOUT INDEX

```
INT .pointer := @i^array[2]; ! pointer variable initialized
                             ! with address of "i^array[2]".
```

If assigned a value through an assignment statement:

```
pointer := 3;
```



puts the value 3 into the word represented by "pointer" (i.e., "i^array[2]).

EQUIVALENCED VARIABLES WITHOUT INDEX

Using an equivalenced variable in a statement or expression treats the memory location as the declared data type of the equivalenced variable.

For example:

```
INT(32) dbl;

INT a = dbl, b = a + 1;
```

If used in an expression

```
a := a * 2;
```

treats the most significant half of "dbl" as type INT.

```
dbl := -1D;
```

treats "dbl" as type INT(32)

USE OF INDEX

Note: The compiler does not perform bounds checking to determine if an <index> value is legitimate. It is possible (and sometimes desirable) to access elements outside of a declared array.

The <index> can be an integer constant:

```
array[3] := n;
```

accesses element 3 of "array"

or a variable:

```
INT array^element^index;                    ! data declaration.

array[array^element^index] := n;
```

the array location accessed depends on the value of "array^element^index".

The <index> can be an arithmetic expression:

```
INT vary;                                   ! data declaration.

array[ vary * 2 ] := ..... ;
```

the "array" element accessed is two times the value of "vary".

```
array[ IF vary = 3 THEN 0 ELSE 6 ] :=  n;
```

the "array" element accessed is conditional depending on the value of "vary".

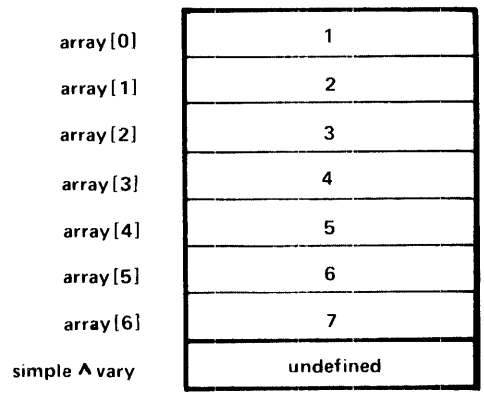
ARRAY VARIABLES WITH INDEX

An individual array element is accessed by appending an <index> to the array identifier.

EXAMPLES OF ARRAY ACCESS WITH INDEX

```
INT array[0:6] := [1,2,3,4,5,6,7],  ! data declarations.
    simple^vary;                    !
```

is allocated as:



An array <variable> can be used in an assignment statement:

```
array[6] := 0;
```

puts the value 0 into "array" element 6.

Or in an arithmetic expression:

```
simple^vary := array[4] + 10;
```

puts the value 15 into "simple^vary".

Or in a conditional expression:

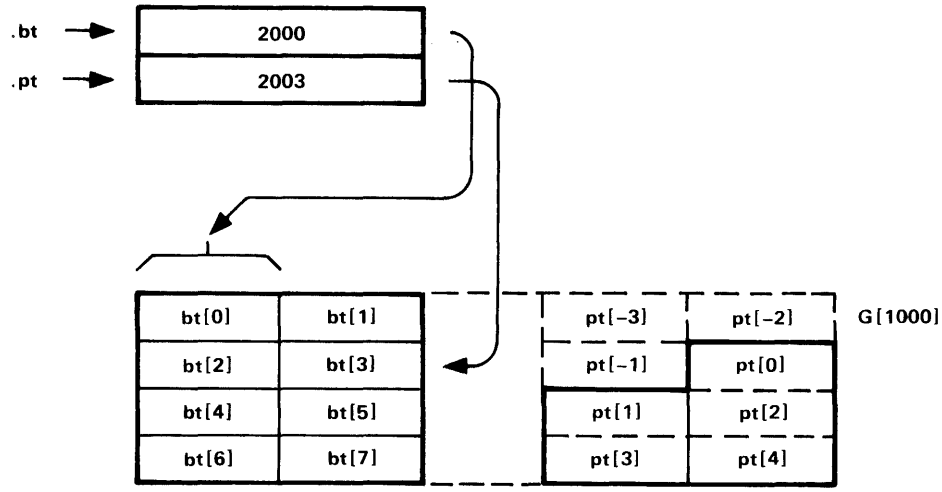
```
IF array[5] = 6 THEN simple^vary := array[2];
```

POINTER VARIABLES WITH INDEX

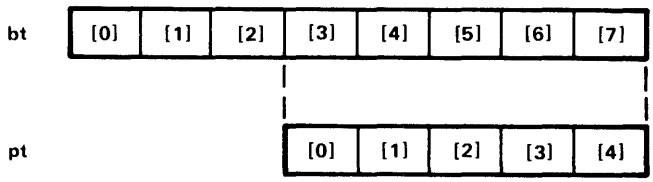
An <index> can be appended to a pointer variable.

EXAMPLE OF POINTER USAGE WITH INDEX

```
    STRING .bt[0:7],                    ! data declarations.
           .pt := @bt[3];               !
```



i.e.



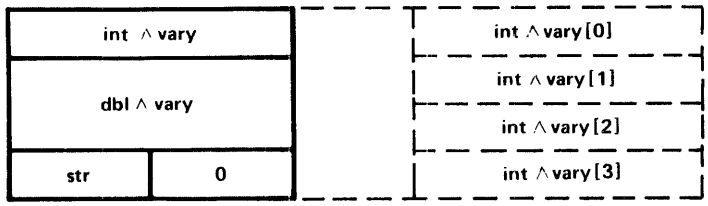

```
    pt[2] := bt;

       puts the value of "bt[0]" into "bt[5]".

 OR

    vary := pt[1] + pt[4];

 is equivalent to

    vary := bt[4] + bt[7];

 OR

    bt[ pt [2] ] := n;

       uses the value in "bt [5]" as an index into the array
       "bt".
```

SIMPLE VARIABLES WITH INDEX

An <index> can be appended to a simple variable.

EXAMPLE OF SIMPLE VARIABLE ACCESS WITH INDEX

```
      INT int^vary;                          ! data declarations.
      INT(32) dbl^vary;                      !
      STRING str;                            !
```



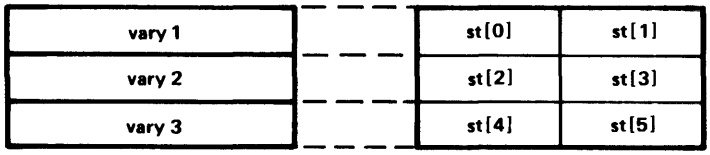

```
      vary := int^vary[1];

         accesses the left half of "dbl^array".

   OR

      vary := int^vary[3];

         accesses "str" in left half of word, 0 in right half.
```

EQUIVALENCED VARIABLES WITH INDEX

An <index> can be assigned to an equivalenced variable.

EXAMPLE OF EQUIVALENCED VARIABLE ACCESS WITH INDEX

```
      INT varyl;                          ! data declarations.
      INT vary2;                          !
      INT vary3;                          !
      STRING st = varyl;                  !
```




```
      st[3] := 0;
```

sets the right half of "vary2" to zero.

OR

```
      IF st[4] = 2 THEN ....
```

checks the left half of "vary3" for the value 2.

A number of arrays could be declared:

```
   INT .a[0:39],                      ! data declarations.
       .b[0:39],                      !
       .c[0:39],                      !
       .d[0:39],                      !
```

A variable could then be equivalenced to the first array:

```
       arrays = a,                    ! data declarations.
      .pointer;                       !
```

"pointer" could be initialized in a statement such as

```
   @pointer := arrays[index];
```

Then accessing

```
   pointer := some^value;
```

puts "some^value" in element [0] of the array specified by "index".

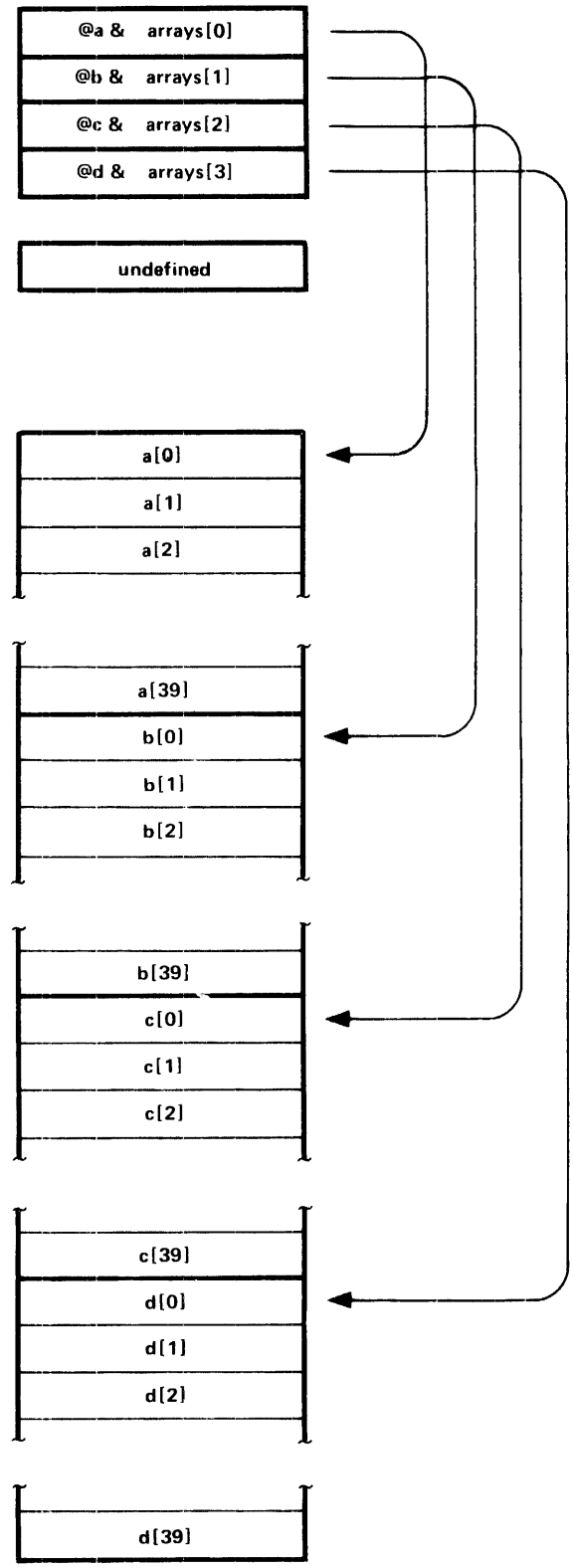
OR

```
   pointer[3] := 0;
```

assigns the value 0 to element [3] of the array specified by "index".

EXAMPLE OF USING EQUIVALENCING TO CONSTRUCT AN ARRAY OF POINTERS

```
INT .a[0:39],
    .b[0:39],
    .c[0:39],
    .d[0:39],
     arrays = a,
    .pointer;
```



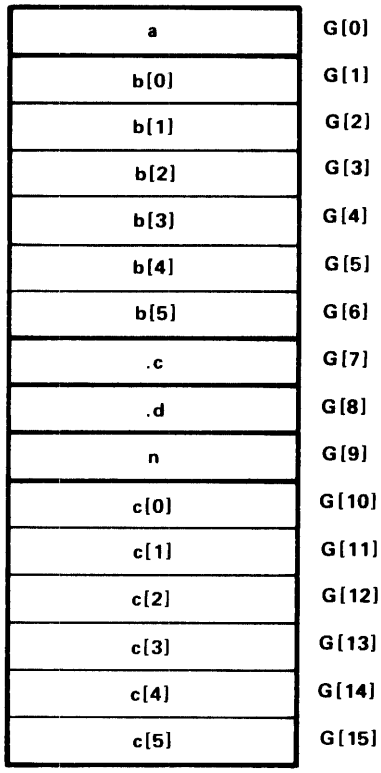
One level of indirection can be removed from a variable by preceding the name of the variable with the symbol for removing indirection (commercial at "@").

The general form is:

```
@ <variable> [ "[" <index> "]" ]
- ----------

where

   @  is the symbol for removing indirection.

   For direct variables (simple variables or direct arrays):

      used in an expression, provides the 'G'[0] relative address
      (plus <index>) of the direct variable <index>) contained in
      the address pointer.  If the direct variable is a STRING
      variable then the address provided is a byte address

   For indirect variables (indirect arrays or pointer variables):

      used in an expression, provides the address (plus <index>)
      contained in the address pointer.  If the the indirect
      variable is a STRING variable then the address provided is
      a byte address

      used on the left side of an assignment statement permits
      modification to the contents of an address pointer

example

   @pointer := @array^name;          ! assignment statement.
```

EXAMPLES OF REMOVING INDIRECTION

```
INT  a,      ! simple.
     b[0:5], ! direct array.
    .c[0:5], ! indirect array.
    .d,      ! pointer
     n;      ! simple
```




```
n := @a;
```

puts the 'G'[0] relative address of "a", 0, into "n".

```
n := @b[3];
```

puts the 'G'[0] relative address of the third element of "b", 4, into "n".

```
n := @c;
```

puts the 'G'[0] relative address of the base of "c", 10, into "n".

```
@d := 1;
```

puts the value 1 into the pointer "d".

```
@d := @c[5];
```

puts the 'G'[0] relative address of the fifth element of "c", 15, into the pointer "d".

```
@a := 1
```

is illegal.

One level of indirection can be specified for a directly addressed variable by preceding the name of the variable with a period ".".

The general form is:

```
. <direct variable> [ "[" <index> "]" ]
- -----------------

where

   .  is the symbol for specifying indirection.  The contents of
      the <direct variable> are used as a 'G'[0] relative address
      pointer.  If the <direct variable> is a STRING variable
      then the contents are used as a byte address

example

   .some^vary := 0;                  ! assignment statement.
```

Preceding a variable with a period "." causes the value of the variable to be used as an address.

```
INT a := 5;                          ! data declaration.

.a := 0;
```

puts the value 0 in cell five of the global area.

A simple variable can be used as a pointer:

```
INT not^pointer;                     ! data declarations.
INT .array [0:7];                    !

not^pointer := @array;
```

puts address of "array" in "not^pointer".

Then

```
.not^pointer := some^value;
```

puts value of "some^value" in element [0] of "array".

OR

```
.not^pointer[1] := some^value + 10;
```

puts value of "some^value" plus 10 in element [1] of "array".

The location actually accessed when a parameter is passed to a procedure or subprocedure and how the variable represented by the parameter is treated depends on the following:

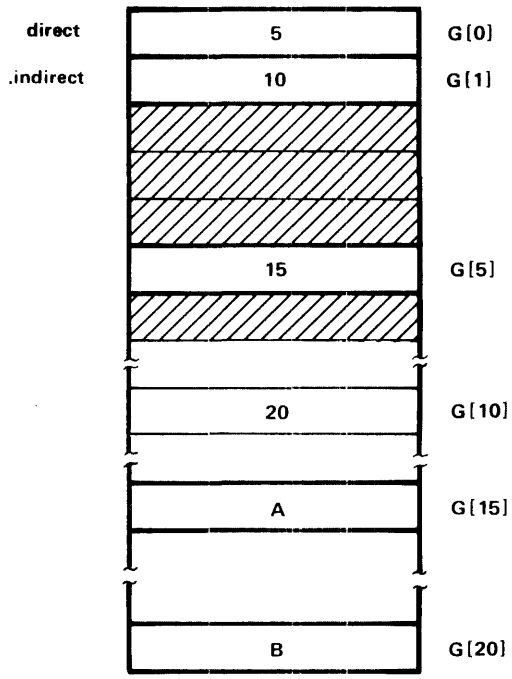
* Whether the parameter was declared by VALUE or by REFERENCE.

* Whether the variable is a direct or indirect variable.

* Whether or not the addressing mode for a variable was changed through use of the symbol for removing indirection or symbol for specifying indirection.

Using the following declarations, the above is clarified:

```
INT  direct := 5,
    .indirect := 10;
```



VALUE PARAMETERS

A value parameter is a copy of an actual variable. The copy is passed in the parameter area and is accessed directly by statements within a procedure.

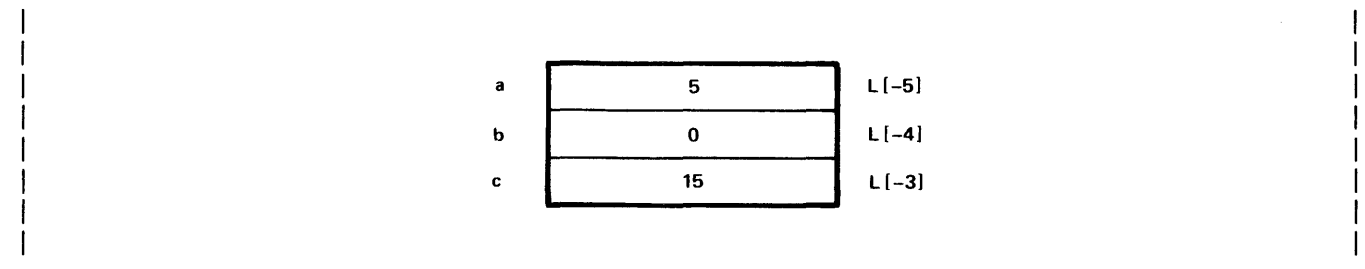
For example:

```
PROC v(a,b,c);
   INT a,b,c; ! value parameters
  BEGIN
    INT n;
    n := a;  n := .a;
    n := b;  n := .b;
    n := c;  n := .c;
  END;
```

If invoked as follows:

```
CALL v(direct, @direct, .direct);
```

results in the following values being passed in the parameter area:



and the local variable "n" taking on the following values:

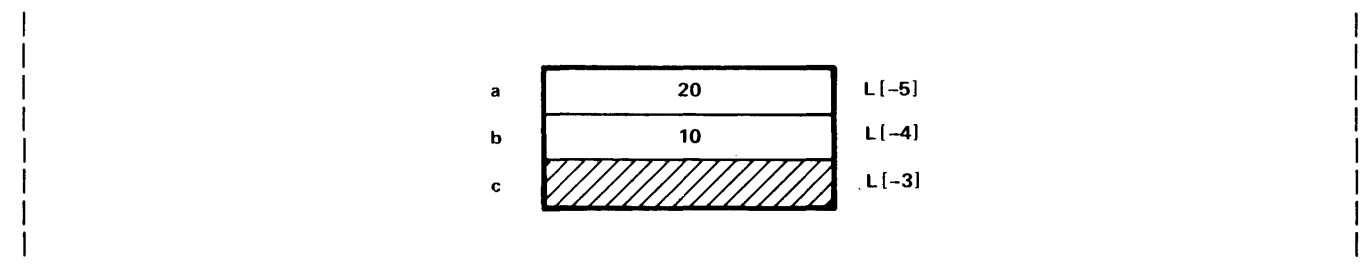
```
for a, "n" is  5; for .a, "n" is  15;
for b, "n" is  0; for .b, "n" is   5;
for c, "n" is 15; for .c, "n" is "A";
```

(@a or @b or @c provides the 'G'[0] relative address of the parameter location)

If invoked with indirect variables as follows:

```
CALL v(indirect, @indirect, .indirect);
                            \_______/
                                |
                            is not permitted
```

results in the following values being passed in the parameter area:



and the local variable "n" taking on the following values:

```
for a, "n" is 20; for .a, "n" is "B";
for b, "n" is 10; for .b, "n" is  20;
```

(@a or @b provide the 'G'[0] relative address of the parameter location)

REFERENCE PARAMETERS

Reference parameters are address pointers to actual variables.

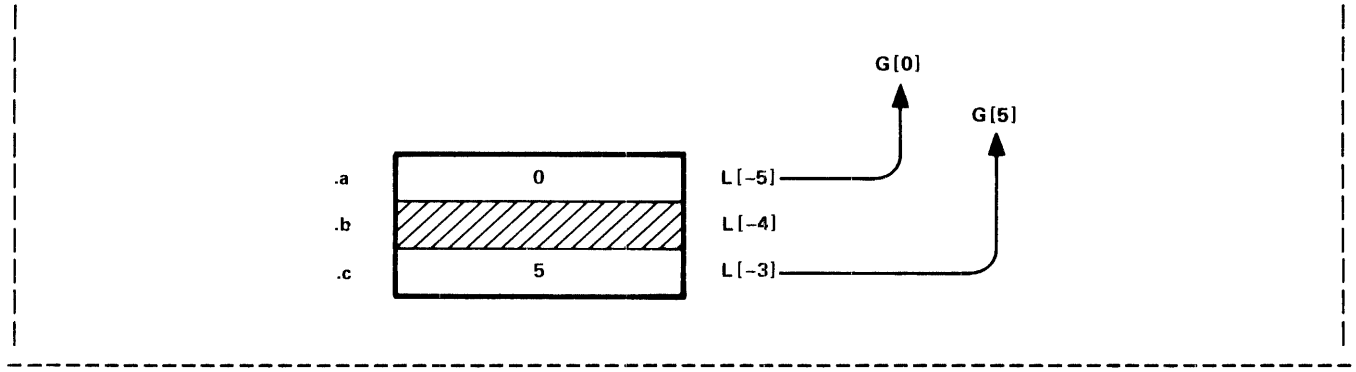
For example:

```
PROC r(a,b,c);
   INT .a,.b,.c; ! reference parameters
  BEGIN
    INT n;
    n := a;  n := @a;
    n := b;  n := @b;
    n := c;  n := @c;
  END;
```

If invoked as follows:

```
CALL r(direct, @direct, .direct);
                \_____/
                   |
                is not permitted
```

results in the following values being passed in the parameter area:



and the local variable "n" taking on the following values:

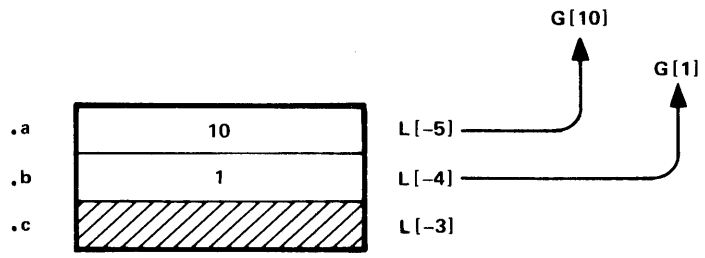
```
for a, "n" is  5; for @a, "n" is 0;
for c, "n" is 15; for @c, "n" is 5;
```

(.a or .b or .c is not permitted)

If invoked with indirect variables as follows:

```
CALL r(indirect, @indirect, .indirect);
                            \_________/
                                 |
                            is not permitted
```

results in the following values being passed in the parameter area:



and the local variable "n" taking on the following values:

```
for a, "n" is 20; for @a, "n" is 10;
for b, "n" is 10; for @b, "n" is  1;
```

Note that the contents of a pointer variable passed as a parameter cannot be changed unless the parameter is declared by reference and the name of the pointer variable is preceded by an at @ symbol at the time of the call.

A T/TAL statement is an order to perform an action. The T/TAL statements are:

	arithmetic
assignment	assigns a value to a variable
	blocking
compound	groups one or more statements into one statement
	transfer of control
GOTO	transfers control to a specific point in a procedure or subprocedure
IF THEN	selects either of two statements for execution depending upon the evaluation of a conditional expression
CASE	selects one of a list of statements for execution depending upon the evaluation of an arithmetic expression
	looping
FOR	repeatedly executes a statement while incrementing a variable until a predetermined limit is reached
WHILE DO	repeatedly executes a statement while a specified condition is true
DO UNTIL	repeatedly executes a statement until a specified condition becomes true
	block (multiple element) operations
move	moves a block of data from one location to another
SCAN	scans a block of data for a particular character
	procedure/subprocedure
CALL	invokes a procedure or subprocedure
RETURN	returns from a procedure or subprocedure to the caller and, if a function procedure or subprocedure, optionally returns a value to the caller

USE OF SEMICOLON TO TERMINATE STATEMENTS

Each statement that is at the outermost level within a procedure or subprocedure body must terminated by a semicolon. Statements that are within other statements are not terminated by a semicolon unless the form of the outer statement requires a terminating semicolon.

The general form of a procedure or subprocedure body is:

```
(SUB)PROC <name> .... ;

   BEGIN

       .
       .
     <statement> ;  <-- these statements are at the outermost level of
     <statement> ;  <-- the (sub)procedure body and, therefore, must
       .          .   .  be terminated by a semicolon.
       .          .   .
       .          .   .
     <statement> ; <--
   END;
```

An example of semicolon usage for a statement of the form

```
   DO <statement> UNTIL <conditional expression>
```

at the outermost level of a procedure body:

```
   PROC myproc;
     BEGIN
         .
         .
       DO a := a + 1 UNTIL a > 10 ;
          ----------
              |                    | terminating semicolon at outer
              |                      level.
              |
              | this assignment statement within the DO .. UNTIL
                statement is not terminated by a semicolon.

     END;
```

The purpose of the assignment statement is to assign a value (the result of an expression) to a variable or part of a variable.

The general form of an assignment statement is:

```
<variable> [ . "<" <left bit> ">" [ : "<" <right bit> ">" ] ]
----------
   := <expression>
   -- ------------

where

   <variable> is a declared data variable with an optional bit
   deposit field

   :=  means "is assigned the value of"

   <expression> represents a value of the same type as <variable>
   (type STRING is treated as type INT)

   Because an assignment statement is a form of arithmetic
   expression, multiple assignments can be performed with a
   single assignment statement

example

   intl := int2 := int3 := vary * 2;
```

Some examples using the following declarations:

```
INT intl, int2;                    ! data declarations.
INT(32) dbll dbl2;                 !
STRING bytel, byte2;               !

intl := intl + 1;

    increments "intl" by one.
```

type STRING is treated as type INT:

```
int2 := bytel * byte2;
```

type mixing is not permitted:

```
dbll := intl;

   is not valid.
```

However, a type transfer function could be used:

```
dbl1 := $DBL( intl );
```

Bit deposit fields may be used with any of the variables:

```
intl.<8:15> := int2.<0:7> := bytel;
```

The variables can be indexed:

```
bytel[8] := byte2;
```

ASSIGNING INT VALUES TO STRING VARIABLES

Caution should be taken when assigning INT values to simple STRING variables and vice-versa:

```
bytel := "AB";
```

results in bytel.<0:7> = "B", bytel.<8:15> = 0.

```
intl := "A";
```

results in intl.<0:7> = 0, intl.<8:15> = "A";

FIXED POINT SCALING IN AN ASSIGNMENT STATEMENT

When a value is assigned to a FIXED variable, the value is scaled up or down as required to match the <fpoint> of the variable. If the value must be scaled down, then some order of precision will be lost. For example:

```
FIXED(2) a;                 ! data declaration.
```

Assigning the value

```
a := 2.345F                 ! FIXED(3) value
```

causes the value to be scaled down one position causing a loss of one digit of precision.

In this example, the value

```
2.34F                       ! FIXED(2) value
```

is stored in "a".

The compiler has the ability to automatically generate instructions for "rounding" a FIXED operand when an assignment to a variable occurs. Rounding is enabled and disabled by the two compiler control commands ?ROUND and ?NOROUND, respectively (see "Compiler Control Commands" for a complete explanation). The default condition, ?NOROUND, causes the value to be truncated if the value must be scaled down prior to the assignment (as shown in the preceding example).

Specifying the ?ROUND compiler command, causes the value to be rounded up, if appropriate, after truncation occurs. For example, specifying the compiler command

```
?ROUND
```

and assigning the value 2.345 to a FIXED(2) variable causes the value to be truncated one digit and rounded up to the value 2.35.

The purpose of the compound statement is to group a number of statements together to form a single statement.

The general form of a compound statement is:

```
BEGIN
-----
  [ [ <statement> ] ; ]
            .
            .
            .
  [ [ <statement> ] ; ]
END
---

where

   <statement>, optional, is any T/TAL statement including other
   compound statements

example

   BEGIN
     integer1 := 0;
     integer2 := IF limit = 0 THEN 1 ELSE 2;
   END;
```

An example of the use of compound statements:

```
BEGIN                                   ! first  compound statement.
  limit := 71;
  IF size > maximum THEN
    BEGIN                               ! second compound statement.
      index := -1;
      WHILE index <= limit DO
        BEGIN                           ! third  compound statement.
          array[index] := index;
          index := index + 1;
        END;                            ! third  compound statement.
    END                                 ! second compound statement.
  ELSE
    BEGIN                               ! fourth  compound statement.
      array := 0;
      index := 0;
    END;                                ! fourth compound statement.
END;                                    ! first  compound statement.
```

Notice that there is no semi-colon ";" on the END terminating the second compound statement (the second compound statement is embedded within the IF statement).

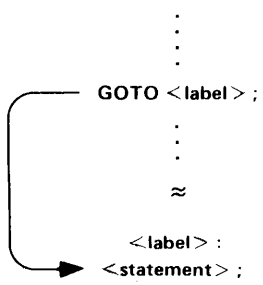
The purpose of the GOTO statement is to unconditionally transfer program control to a "labeled" statement.

The form of the GOTO statement is:

```
GOTO <label>
---- -------

where the form of the label is:

   <label>:  <statement>;

   where

      <label> is an identifier, followed by a colon ":",
      indicating the statement to be executed as a result of the
      GOTO statement

example

   GOTO start;
```

A label is the only element in T/TAL that need not be declared before being used.

GOTO STATEMENT ACTION



Any <statement> can be labeled:

```
IF a > b THEN <statement> ELSE <label>: <statement> ;
```

the following is invalid (ELSE does not start a <statement>):

```
IF a > b THEN <statement> <label>: ELSE ...;
```

A GOTO cannot be used to leave a procedure, but can be used to leave a subprocedure back to the calling procedure (but not to another subprocedure).

For example:

```
PROC main^proc MAIN; BEGIN
      .
      .
    SUBPROC search (a);
       INT .a;
      BEGIN
          .
        IF NOT found THEN
          IF NOT error THEN CALL search(a)
          ELSE GOTO main^; ! error occurred, bail out.
          .

           calls "a" recursively if "error" is false, otherwise
           goes directly back to the where "a" was invoked
           originally (i.e., does not have to make return through
           the nested calls)

      END;

          is the normal return (i.e., back through the nested
          calls).

      .
    CALL search(list);

          starts the recursive calls to "search".

    main^:

          is a label for the error return from "search"
```

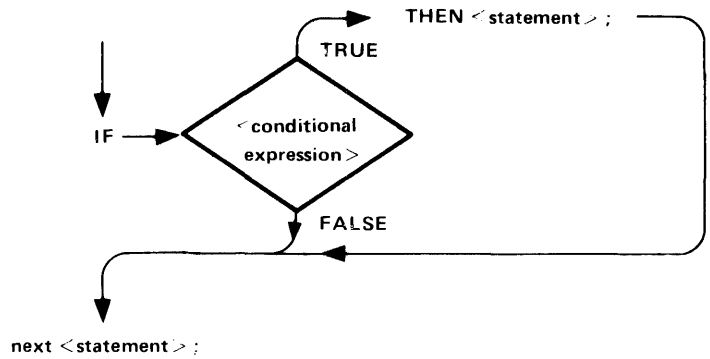
The purpose of the IF statement is to permit the state of a conditional expression (ie, true or false) to determine which of two statements is to be executed.

The general form of the IF statement is:

```
IF <conditional expression> THEN [ <statement> ]
-- ------------------------ ----

  or

IF <conditional expression> THEN [ <statement> ]
-- ------------------------ ----
ELSE [ <statement> ]
----

where

   <statement> ,optional, is any T/TAL statement including
   compound statements and IF statements

example

   IF number = limit THEN number := 0;
```

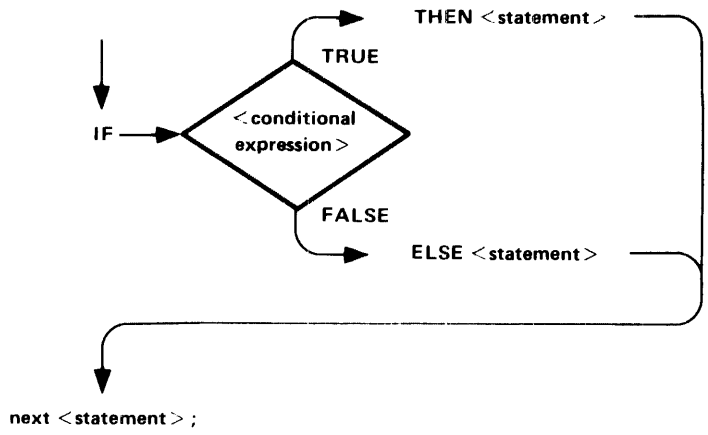
Using the form without the ELSE part, the <statement> following THEN is executed if the condition is true, otherwise it is skipped:

IF STATEMENT ACTION (IF THEN FORM)



Using the form with the ELSE part, the <statement> following THEN is executed if the condition is true, otherwise the <statement> following ELSE is executed:

IF STATEMENT ACTION (IF THEN ELSE FORM)



Some examples:

```
IF item^num = taxable THEN tax := compute^tax;

IF cost >= limit THEN
  BEGIN
    bad^item := 1;
    item^count := 0;
  END
ELSE item^count := item^count + 1;
```

IF statements can be nested indefinitely:

```
IF <conditional expression> THEN
  IF <conditional expression> THEN
    IF <conditional expression> THEN
      IF <conditional expression> THEN
        BEGIN                                ! compound statement.
          IF <conditional expression> THEN <statement>;
        END                                  ! compound statement.
      ELSE <statement>
    ELSE <statement>
  ELSE <statement>;
                                   ! outermost ELSE not required.
```

The innermost THEN is paired with the closest ELSE and pairing proceeds outward. However, compound statements can be used to override pairing.

An IF statement can be used to check a variable for a non-zero state:

```
IF invalid^item THEN CALL ABEND;
```

checks "invalid^item" for non-zero value.

An IF statement can be used to check the hardware condition code setting:

```
CALL READ(filenum,....);
IF < THEN
  BEGIN
    print^errmsg;
    CALL STOP;
  END;
```

is a call to the file system READ Procedure. The condition code is set to less-than if an error occurs.

An IF statement can be used to compare two arrays:

```
IF in^array = mask^array FOR 25 THEN item^found := 1
ELSE item^found := 0;
```

compares "in^array" to "mask^array" for 25 elements.

```
IF file^name = "$RECEIVE" THEN this^message := cpu^pin;
```

The <statement> parts of the IF statement are optional. For example:

```
IF not^first^time THEN ELSE
  BEGIN                 ^
    .                   |
    .                   omitted statement.
  END;
```

The purpose of the CASE statement is to selectively execute one of a list of statements, the statement executed being determined by an index value.

The general form of the CASE statement is:

```
CASE <index> OF
---- ------- --
  BEGIN
  -----
    [ <statement for index = 0> ] ;
                                  -
    [ <statement for index = 1> ] ;
                                  -
        .
        .
        .
    [ <statement for index = n> ] ;
                                  -
    [ OTHERWISE [ <statement> ] ; ]
  END
  ---
```

where

<index> is an arithmetic expression indicating the statement number of the <statement> to be executed

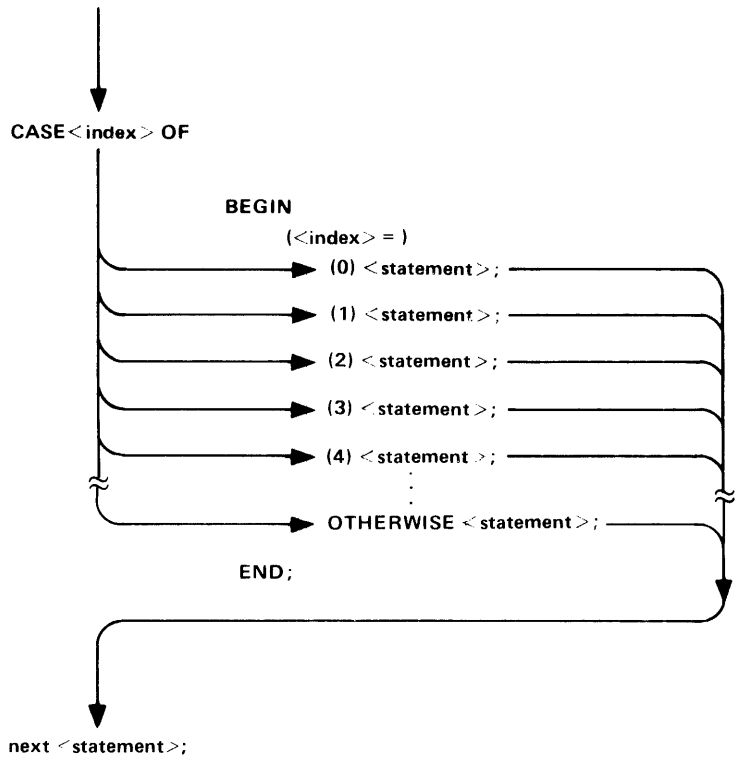
<statement>, optional, is any T/TAL statement including compound statements and CASE statements

OTHERWISE, optional, indicates an alternate (optional) statement to be executed if <index> does not point to one of the statements in the CASE body

example

```
CASE some^variable OF
  BEGIN
    vary0 := 0;
    varyl := 1;
    OTHERWISE CALL DEBUG;
  END;
```

CASE STATEMENT ACTION



Unless the OTHERWISE <statement> is used, the value of <index> must point to one of the statements in the CASE compound body. Otherwise, the results will be unpredictable.

The <statement> parts of the CASE statement are optional. A semicolon holds the place (i.e., indicates the index) of an omitted statement. For example:

```
CASE index OF
  BEGIN
    ;              <----  placeholder, no 0'th index value or no
                          action to be taken.
    CALL a;        <----  1'st index value.
    CALL b;        <----  2'nd index value.
    OTHERWISE ;
  END;       ^
             |
```

omitted statement. However, the use of "OTHERWISE" means the a value of "index" that is greater than two has a predictable result.

The purpose of the FOR statement is to repeatedly execute a <statement> while stepping a variable until that variable exceeds a specified limit.

The FOR statement provides an efficient means of indexing through array elements.

The general form of the FOR statement is:

```
                                 { TO     }
FOR <variable> := <initial> { DOWNTO } <limit> [ BY <step> ]
--- ---------- -- ---------   ------   -------
   DO [ <statement> ]
   --
```

where

<variable> is incremented or decremented by the <step> value each time the <statement> is executed

<initial> is an arithmetic expression specifying an initial value to be assigned to <variable>

TO means increment the <variable> by the <step> value

DOWNTO means decrement the <variable> by the <step> value

<limit> is a variable or an arithmetic expression that is compared to <variable>. If <variable> is less than or equal to <limit>, the <statement> following DO is executed. When <variable> exceeds limit, the FOR statement is finished

<step> is a variable or an arithmetic expression specifying a positive amount <variable> is to be incremented (TO) or decremented (DOWNTO) each time the <statement> is executed. If <step> is not included, one (1) is assumed

<statement>, optional, is any T/TAL statement including compound statements and FOR statements

example

```
FOR index := 0 TO length - 1 DO array[index] := " ";
```

Execution of the FOR statement proceeds as follows:

* When the FOR statement is initially entered, the value of <initial> is calculated and stored in <variable>. If <limit> and <step> are arithmetic expressions, their values are also calculated.

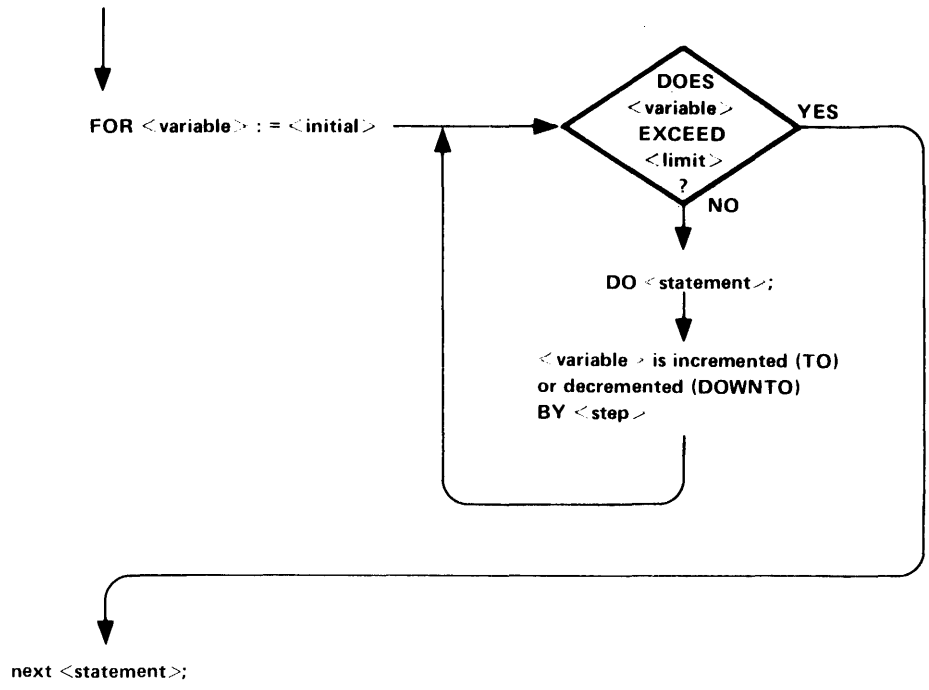
* <variable> is then tested to check whether it exceeds the <limit> value. If not, the <statement> following DO is executed.

* After <statement> is executed, the <step> value is added to (TO) or subtracted from (DOWNTO) <variable> and compared with the <limit> value. If <variable> does not exceed <limit>, the <statement> is again executed.

* If <variable> exceeds <limit>, <statement> is not executed and program execution falls through to the statement following the FOR statement.

If the TO form of the FOR statement is used, <variable> exceeds <limit> when it is more positive than <limit>. If the DOWNTO form is used, <variable> exceeds <limit> when it is more negative than <limit>.

FOR STATEMENT ACTION



The BY part can be omitted providing an implicit <step> of 1:

```
FOR index := 0 TO length-1 DO
  IF array[index] <> " " THEN last^non^blank := index;
```

BY part included (in DOWNTO form):

```
FOR index := length - 1 DOWNTO 1 BY 2 DO
  IF array[index] = " " THEN first^non^blank := index;
```

Using a compound statement:

```
FOR vary := 0 TO num^items BY 2 DO
  BEGIN
    out^array[index] := in^array[vary];
    index := IF out^array[index] = special^case THEN 0
             ELSE index +1;
  END;
```

Using a nested FOR statement:

```
FOR outer^var := 0 TO 20 DO
  FOR inner^var := 0 TO 20 DO <statement>;
```

<statement> is executed 21 times each time the inner FOR statement is executed (total of 441 times).

Some restrictions to the use of the FOR statement:

* FOR statements should be entered only from the beginning. Never branch into the loop <statement>.

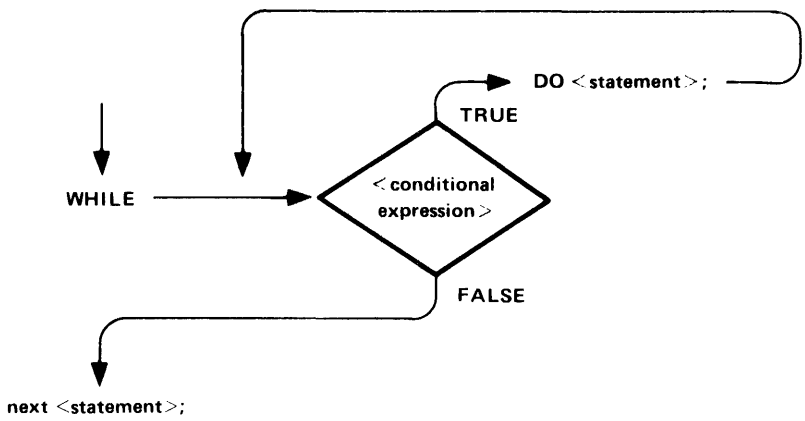
The purpose of the WHILE statement is to repeatedly execute a statement as long as a specified condition is true.

The general form of the WHILE statement is:

```
WHILE <conditional expression> DO [ <statement> ]
----- ------------------------ --

where

   <statement>, optional, is any T/TAL statement including
   compound statements and WHILE statements

example

   WHILE number < limit DO number := number + 1;
```

The <conditional expression> is evaluated and tested before the <statement> is executed. When the condition becomes false, program execution continues with the statement following the WHILE statement:

WHILE STATEMENT ACTION



The <statement> can be a compound statement:

```
WHILE NOT alpha^found DO
  BEGIN
    IF $SPECIAL(array[index]) THEN index := index + 1;
    IF = THEN alpha^found := 1;
      .
      .
  END;
```

A WHILE statement can be used to indefinitely execute a statement:

```
WHILE 1 DO
  BEGIN
  <statement>;
    .
    .
  END;
```

the compound statement will be executed indefinitely because 1 is always evaluated to a non-zero value.

The <conditional expression> part of the WHILE statement can be a function procedure:

```
WHILE cmd^num := find^command DO
  CASE cmd^num OF
    BEGIN
      <statement>;
      <statement>;
           .
           .
      OTHERWISE GOTO abort^loop;
    END;
```

executes as long as "cmd^num" is a non-zero value.

The <statement> part of the WHILE statement is optional. For example:

```
index := -1;
WHILE $ALPHA(array[index := index + 1]) DO;
                                        ^
                                        |
                                        omitted statement.
```

increments "index" until a non-alphabetical character is referenced.

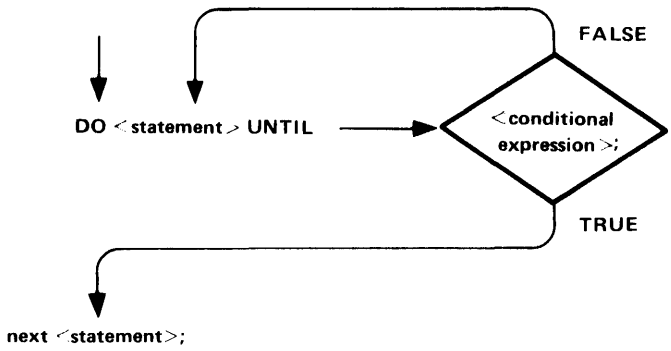
The purpose of the DO statement is to repeatedly execute a statement until a specified condition becomes true.

The general form of the DO statement is:

```
DO [ <statement> ] UNTIL <conditional expression>
--                 ----- -----------------------

where

   <statement>, optional, is any T/TAL statement including
   compound statements and DO statements

example

   DO array[index := index + 1] := 0 UNTIL  index >= limit;
```

The <statement> is executed before the <conditional expression> is evaluated, therefore <statement> is always executed at least once. When the condition becomes true, program execution continues with the statement following the DO statement.

DO STATEMENT ACTION



An example:

```
DO index := index + 1 UNTIL $ALPHA(array[ index ]);
```

The purpose of the move statement is to move a block of information from a source array or a constant list into a destination array. When a move is executed, elements are moved into the destination array, one element at a time, in the direction specified by the move operator.

The general form of the move statement is:

```
                { ':=' } { <s array> FOR <number of elements> }
<d array>       { '=:' } { <constant>                         }
---------         ----     -------------------------------
   [ -> <next address> ]
```

where

<d array> is the name of the destination array

':=' is a left-to-right move operator
'=:' is a right-to-left move operator

<s array> is the name of the source array

<number of elements> is an <arithmetic expression> of the general form specifying the maximum number of elements in <s array> to be moved. <number of elements> is treated as an unsigned value thereby permitting {0:65535} elements to be moved in a single operation

<next address> is a variable that is assigned an address following completion of the move. The address points to the next location in <d array> following the last element moved. If <d array> is a word-addressed variable, then this is a 'G'[0]-relative word address; if <d array> is a byte-addressed variable, then this is a 'G'[0]-relative byte address

several "<s array> FOR <elements>" and/or "<constant>" can be combined using ampersands "&" to provide concatenating moves

Moves between word-addressed variables of different data types (e.g., INT and INT(32)) are permitted. The number of words moved is dependent on the data type of <s array>

For byte-addressed variables, both <d array> and <s array> must be byte-addressed or <s array> must be a constant

example

```
out^array ':=' in^array FOR 72;
```

Note: When using an integer constant or a literal identifier in a byte move operation, the constant/identifier should be

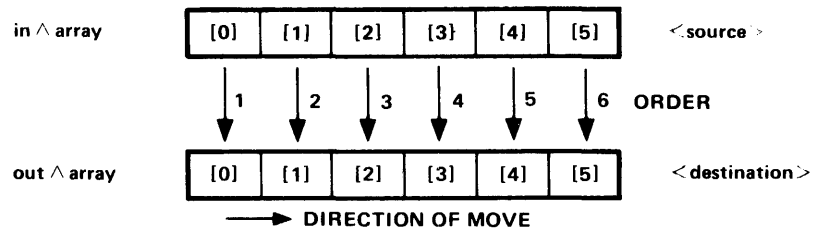
surrounded by brackets "[...]" if a one-byte value is desired. If a constant value is not surrounded by brackets, it is treated as two bytes in a byte move operation.

LEFT-TO-RIGHT MOVE

A left-to-right move starts with low order addresses in the <s array> and <d array> arrays and increments the addresses as the move progresses (i.e., picks up elements from left-to-right).

EXAMPLE OF LEFT-TO-RIGHT MOVE

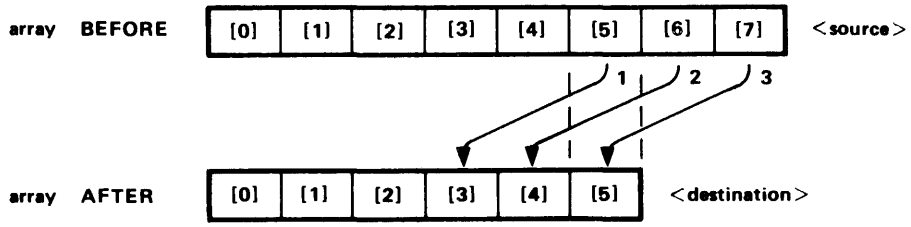
```
out^array ':=' in^array FOR 6;
```



A left-to-right move can be used for a deletion operation. For example, it is desired to delete two elements from an array containing eight elements, starting with element [3]:

EXAMPLE OF LEFT-TO-RIGHT MOVE TO PERFORM DELETION

```
INT array[0:7];

array[3] ':=' array[5] FOR 3;
```

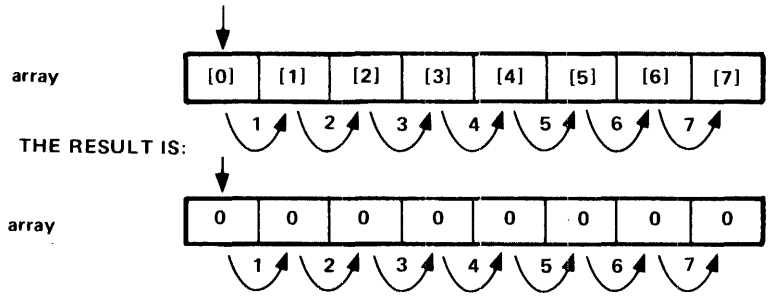


moves "array[5:7]" over "array[3:5]"

A left-to-right move can also be useful for initializing an array.

EXAMPLE OF LEFT-TO-RIGHT MOVE FOR INITIALIZING AN ARRAY

```
array := 0;
array[1] ':=' array FOR 7;
```



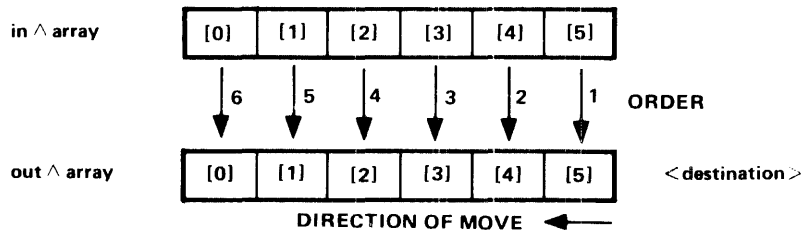
fills eight elements of "array" with zeros

RIGHT-TO-LEFT MOVE

A right-to-left move starts with high order addresses in the <s array> and <d array> arrays and decrements the addresses as the move progresses (i.e., picks up elements from right-to-left):

EXAMPLE OF RIGHT-TO-LEFT MOVE

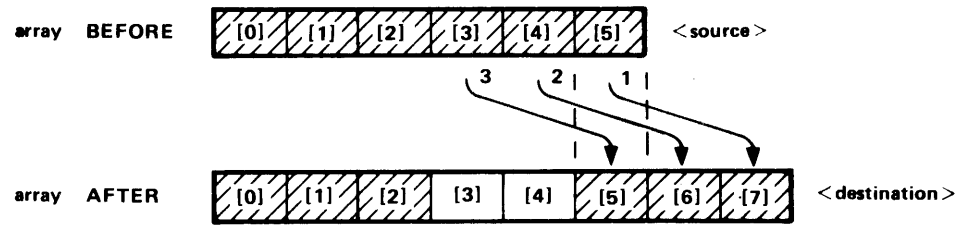
```
out^array[5] '=:' in^array[5] FOR 6;
```



A right-to-left move can be used for an insertion operation. For example, two elements are to be inserted in a eight-element array starting with element[3].

EXAMPLE OF RIGHT-TO-LEFT MOVE TO PERFORM INSERTION

```
    array[7] '=:' array[5] FOR 3;
```



provides space for two elements, starting with "array[3]"

USING <next address>

<next address> provides the 'G'[0] relative address of the next location in memory after the last item moved.

```
STRING .pointer,                    ! data declarations.
       .in^array[0:71],             !
       .out^array[0:71],            !
       .join^array[0:71];           !
```

then performing a left-to-right move

```
join^array ':=' in^array FOR 4 -> @pointer;
```

results in "pointer" equal to "@join^array[4]"

Then subtracting

```
num^moved := @pointer '-' @join^array;
```

results in "num^moved" = 4 (four elements moved)

"pointer" could then be used as the <d array> array:

```
pointer ':=' out^array FOR 4;
```

"join^array" would then consist of four elements of "in^array" followed by four elements of "out^array"

Similar results are obtained if a right-to-left move is performed:

```
join^array[10] '=:' in^array[71] FOR 6 -> @pointer;
```

results in "pointer" equal to "@join^array[4]"

Then subtracting

```
num^moved := @join^array[10] '-' @pointer;
```

results in "num^moved" = 6 (six elements moved)

Another move could then be performed as follows:

```
pointer '=:' out^array FOR @pointer '-' @join^array;
```

"join^array" would then consist of four elements of "out^array" followed by six elements of "in^array"

USING <constant>

```
file^name ':=' [ "$RECEIVE", 8 * ["  "]] -> @next^addr;
```

"file^name" then contains the constant list, "next^addr" points to "file^name[12]"

subtracting

```
num^moved := @next^addr '-' @file^name;
```

results in "num^moved" = 12 (twelve elements)

USING CONCATENATING MOVES

A number of arrays and/or constant lists can be joined together using a concatenating move.

```
INT varyl := 2;
STRING date [0:10] := "MAY 1, 1976";
STRING account^num [0:10] := "123-456-789";
STRING amount [0:10] := "24.99";
```

then performing the left-to-right move

```
print^line ':=' "DATE:  " & date FOR varyl * 5 + 1 & "  ACCT NO:  "
                & account^num FOR 11 & "  ***$"
                & amount FOR varyl * 2 + 1 -> @pointer;
```

results in "print^line" containing;

```
"DATE:  MAY 1, 1976  ACCT NO:  123-456-789  ***$24.99"
```

and subtracting

```
num^moved := @pointer '-' @print^line;
```

provides the number of elements moved into "print^line"

The purpose of the scan statement is to search a STRING array for a particular character.

The general form of the scan statement is:

```
{ SCAN  }          { WHILE }
{ RSCAN } <array>  { UNTIL } <test character> [ -> <next address> ]
  -----    -------   -----    ----------------
```

where

SCAN indicates a left-to-right scan

RSCAN indicates a right-to-left scan

<array> is the <type> STRING array to be scanned

WHILE indicates that the scan continues while the <test character> is found

UNTIL indicates that the scan continues until the <test character> is found

A scan will terminate if an ASCII null (i.e., 0) character is encountered

<test character> is an <arithmetic expression> of the general form resulting in no more than eight significant bits

<next address> is a variable that will contain an address following completion of the scan. If a scan WHILE was performed, <next address> points to the first character that did not match. If a scan UNTIL was performed, <next address> points to the first character that matched. <next address> will, in either case, point to any terminating null character

The SCAN statement also sets the hardware CARRY bit if the scan was terminated because a null character was found

example

```
SCAN in^array WHILE " " -> @first^non^blank;
```

Any array that is to be scanned should terminate, on the opposite end from the start of the scan, with an ASCII null character (i.e., 0).

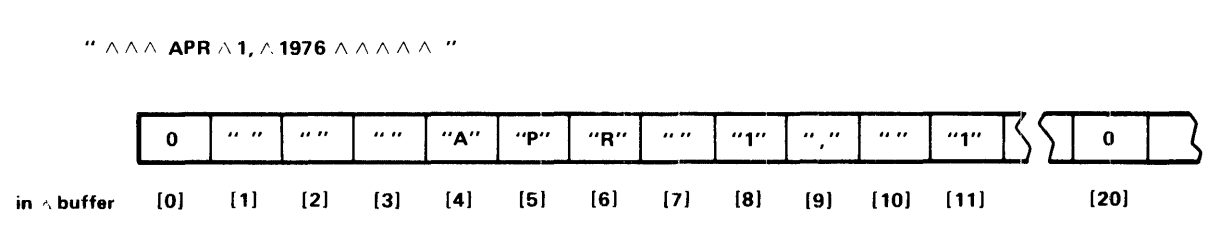
For the following examples, assume that the following record was placed in a byte array "in^buffer":

```
"    APR 1, 1976   ";
```

and the "in^buffer" was delimited with a null character as follows:

```
in^buffer := in^buffer[20] := 0;
```

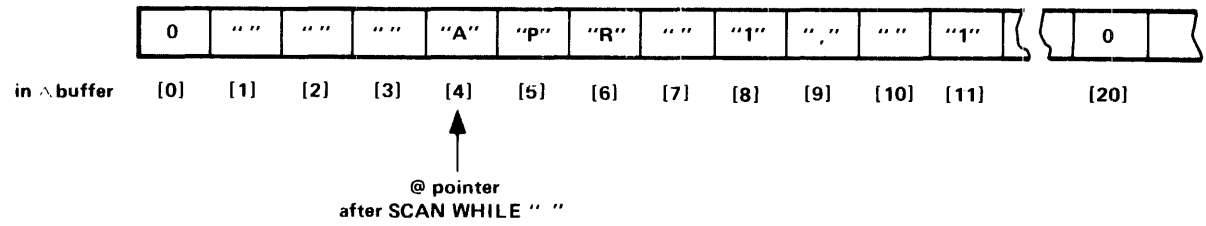
resulting in



SCAN WHILE

EXAMPLE OF LEFT-TO-RIGHT SCAN WHILE

```
SCAN in^buffer[1] WHILE " " -> @pointer;
```



The scan stops when the first non-blank character (i.e., "A") is found. The address of the first non-blank is stored in "pointer" (i.e., "@in^buffer[4]"):

Then the relative location of "A" can be computed as follows:

```
month := @pointer '-' @in^buffer;
```

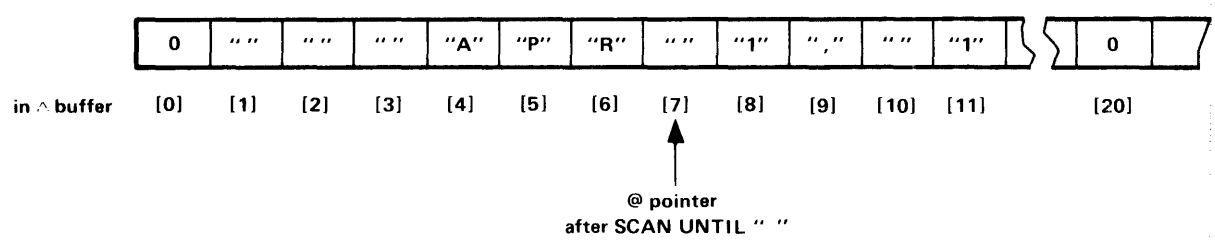
four.

SCAN UNTIL

Using the value of "pointer" that resulted from the preceding example, another left-to-right scan could be made to determine the length of the first item:

EXAMPLE OF LEFT-TO-RIGHT SCAN UNTIL

```
SCAN pointer UNTIL " " -> @pointer;
```



The scan stops when the next blank is found. The address of the next character is stored in "pointer" (i.e., "@in^buffer[7]"):

Then the length can be computed as follows:

```
month^length := @pointer - @in^buffer[month];
```

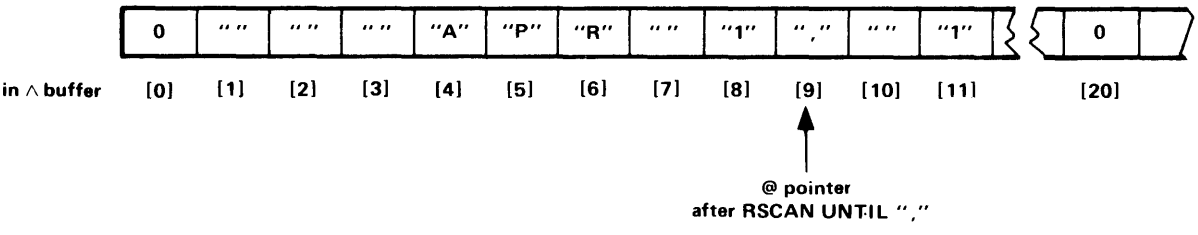
three characters.

RSCAN UNTIL

Again using the original declarations and performing a right-to-left scan:

EXAMPLE OF RIGHT-TO-LEFT SCAN UNTIL

```
INT test^word;                          ! data declaration.
  .
test^word := ",";
RSCAN in^buffer[19] UNTIL test^word -> @pointer;
```



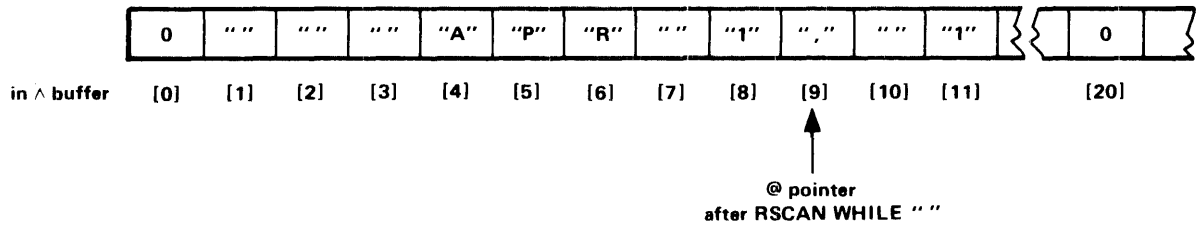
The scan stops when the character "," is found. Then the address where "," is located is stored in "pointer" (i.e., "in^buffer[9]").

RSCAN WHILE

Using the contents of "pointer" from the preceding example and performing the following right-to-left scan

EXAMPLE OF RIGHT-TO-LEFT SCAN WHILE

```
RSCAN pointer WHILE " " -> @pointer;
```



In this case, the scan stops immediately (i.e., on the first character). The address stored in "pointer" is the same as the starting address (i.e., "@in^buffer[9]").

CHECKING CARRY

If the CARRY indicator is to be checked, it should be checked before performing any arithmetic operations (arithmetic operations change the state of the CARRY indicator).

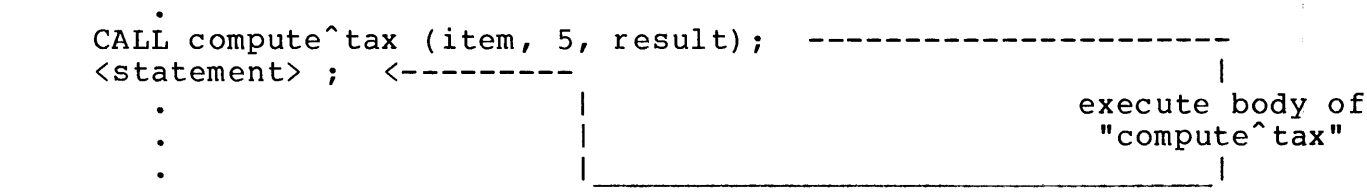
```
IF $CARRY THEN ... ;                ! test character not found.

IF NOT $CARRY THEN                  ! test character found.
```

The purpose of the CALL statement is to execute a procedure or subprocedure.

The general form of the CALL statement is:

```
CALL <name>
---- ------

CALL <name> ( <param 1> , <param 2> , ... <param n> )
---- ------ - --------- - --------- -     --------- -

where

   <name> is the name given to a previously declared procedure or
   subprocedure

   <param> is the actual parameter to be passed to <name> for a
   corresponding <formal parameter name>.  If the formal
   parameter is a value parameter, either a <variable> or an
   <expression> can be passed.  If the formal parameter is a
   reference parameter, a <variable> must be passed

   Unless <name> has been declared as having optional parameters,
   a <param> must be supplied for each formal parameter that was
   declared

   If <name> has optional parameters, omitted parameters are
   indicated by no entry in the corresponding <param> position.
   However, placeholder commas "," must be included (except for
   rightmost missing parameters)

example

  CALL compute^tax(item, 5, result);
```

When a procedure or subprocedure is invoked, the body of the procedure is executed, program execution then returns to the statement following the CALL statement:



The CALL statement can be used to call function procedures. However, the value assigned to the procedure or subprocedure is lost.

An example of calling a procedure with optional parameters:

```
CALL FILEINFO (filenum, err^num,,dev^num,,,eof);
```

Note: The compiler has no way of knowing which parameters are required by a procedure having optional parameters. This checking must be performed by the called procedure itself.

The RETURN statement has two functions: to provide additional points within the procedure or subprocedure body to exit back to the caller and additionally, to return a value from a function procedure or subprocedure. Note that, if a procedure is designated MAIN, program execution stops if a RETURN statement is encountered.

The two forms of the RETURN statement are:

```
RETURN                                  (simple form)

   and

RETURN <expression>                     (expression form)
------ ------------

where

   <arithmetic expression> is a value of the same type as the
   procedure or subprocedure

example

   RETURN local^vary;                   ! expression form.
```

Some examples using the simple form of the RETURN statement:

```
PROC some^proc;

  BEGIN
    .
    IF a < b THEN RETURN;               ! returns to caller.
      .
      .
    RETURN;
      .
  END;
```

OR

```
PROC main^proc MAIN;

  BEGIN
    .
    IF end^of^program THEN RETURN;      ! program execution stops.
      .
      .
  END;                                  ! program execution stops.
```

An example of using the expression form of the RETURN statement. An INT type subprocedure is declared:

```
INT SUBPROC find^record (acct^no);
   INT acct^no;

     BEGIN

       INT temp^store;
         .
         .
       temp^store := acct^no * 10;
         .
         .
       RETURN temp^store;              ! returns to caller.
     END;
```

Then used in an expression as follows:

```
IF ( read^this := find^record(account^no)) <> 0 THEN ......
```

As part of T/TAL, a number of commonly used functions are provided. These include:

* Type Transfer functions (treats a variable of one data type as another data type)

* Character Test functions (tests an ASCII character for being an alpha, numeric, or special (ie, not alpha or numeric).

* MIN/MAX functions (provides the minimum or maximum value of two expressions)

* Carry and Overflow tests (tests the hardware Carry and Overflow indicators)

* Fixed point functions scale and point (scales a fixed operand and returns the <fpoint> value of an expression)

The standard functions are <primarys> (see "Expressions").

Note: Using a standard function does not alter the original contents of a variable specified as a parameter to the function.

T/TAL does not permit data type mixing within expressions. The type transfers functions are used when computing values involving more than one data type.

The type transfer functions are:

$INT (<dbl expression>)

returns an INT from the right half of an INT(32)

$HIGH (<dbl expression>)

returns an INT from the left half of an INT(32)

$DBLL (<int expression> , <int expression>)

returns an INT(32) from two INTs

$DBL (<int expression>)

returns a signed INT(32) from a signed INT (ie, performs a signed right shift, 16 positions)

$UDBL (<int expression>)

returns an INT(32) from an unsigned INT (left half of INT(32) set to 0's)

$COMP (<int expression>)

returns the one's complement of an INT

$ABS (<int expression>)

returns the absolute value of an INT

$IFIX (<int expression> , <fpoint>)

returns a 64-bit integer from a signed integer expression (i.e., performs the equivalent to a signed right shift of 48 positions). $IFIX is treated as having an assumed decimal position of <fpoint>.

$LFIX (<int expression> , <fpoint>)

returns a 64-bit integer from an unsigned integer expression (i.e., the unsigned integer is put in the least significant word of the quadrupleword; the three most significant words are set to zero). $LFIX is treated as having an assumed decimal position of <fpoint>

-->

$DFIX (<dbl expression> , <fpoint>)

returns a 64-bit integer from a signed double word integer expression (i.e., performs the equivalent to a signed right shift of 32 positions). $DFIX is treated as having an assumed decimal position of <fpoint>

$FIXI (<fixed expression>)

returns the signed integer equivalent of a fixed expression. The fixed expression is treated as a 64-bit integer; an (implied) decimal point is ignored. $FIXI causes Overflow to be set if the result cannot be represented in a signed integer (i.e., 15 bits)

$FIXL (<fixed expression>)

returns the unsigned integer equivalent of a fixed expression. The fixed expression is treated as a 64-bit integer; an (implied) decimal point is ignored. $FIXL causes Overflow to be set if the result cannot be represented in an unsigned integer (i.e., 16 bits)

$FIXD (<fixed expression>)

returns the INT(32) equivalent of a fixed expression. The fixed expression is treated as a 64-bit integer; an (implied) decimal point is ignored. $FIXD causes Overflow to be set if the result cannot be represented in a signed doubleword integer (i.e., 31 bits)

where

<int expression> is an arithmetic expression giving an INT result.

<dbl expression> is an arithmetic expression giving an INT(32) result

<fixed expression> is an arithmetic expression giving a FIXED result

<fpoint> is the number of positions that the implied decimal point is to the left (<fpoint> > 0) or to the right (<fpoint> <= 0) of the least significant digit. <fpoint> is represented by an integer constant, the range of which is -19 to + 19

example

```
some^int := $INT(some^double);
```

Some examples using the following declarations:

```
INT some^int;                          ! data declaration.
INT(32) some^double;                   !
FIXED(3) some^fixed;                   !

INT PROC int^proc;                     ! procedure declaration.
  BEGIN
    RETURN 2 * some^int;
  END;
```

$INT

```
some^int := $INT(some^double);
```

assigns the value of right half of "some^double" to some^int.

or

```
IF $INT(some^double) = %255 THEN .....;
```

checks right half of "some^double" for equality with %255.

$HIGH

```
some^int := $HIGH(some^double);
```

assigns the value of left half of "some^double" to "some^int" or

```
IF $HIGH(some^double) THEN...;
```

checks left half of "some^double" for nonzero value.

$DBLL

```
some^double := $DBLL(some^int, intproc);
```

assigns the value of "some^int" to left half of "some^double", the value returned from "int^proc" to the right half.

$DBL

```
   INT some^int := -1;                    ! data declaration.
```

then

```
   some^double := $DBL(some^int);
```

assigns the value of "some^int" to right half of "some^double", propagating the sign bit.

results in the 32-bit quantity (-1D):

```
                                        1 1 1  1 1 1
            0  1 2 3  4 5 6  7 8 9  0 1 2  3 4 5

   left half of "some^double"

            1  1 1 1  1 1 1  1 1 1  1 1 1  1 1 1

   right half of "some^double"

            1  1 1 1  1 1 1  1 1 1  1 1 1  1 1 1
```

or

```
   some^double := some^double + $DBL(int^proc);
```

converts the value returned from "int^proc" to an INT(32) then adds that value to "some^double".

$UDBL

```
   INT some^int := -1;                    ! data declaration.
```

then

```
   some^double := $UDBL(some^int);
```

assigns the value of "some^int" to the right half of "some^double", the left half is filled with zeros.

results in "some^double" = %177777D

$COMP

```
some^int := $COMP(10);
```

assigns the one's complement of 10 to "some^int".

```
                                  1 1 1  1 1 1
      0  1 2 3  4 5 6  7 8 9  0 1 2  3 4 5
```

10 is equivalent to:

```
      0  0 0 0  0 0 0  0 0 0  0 0 1  0 1 0
```

then complementing 10 results in:

```
      1  1 1 1  1 1 1  1 1 1  1 1 0  1 0 1
```

$ABS

```
some^int := $ABS(-2);
```

assigns the absolute value of -2 to "some^int".

```
                                  1 1 1  1 1 1
      0  1 2 3  4 5 6  7 8 9  0 1 2  3 4 5
```

-2 is equivalent to:

```
      1  1 1 1  1 1 1  1 1 1  1 1 1  1 1 0
```

then the absolute value of -2 is:

```
      0  0 0 0  0 0 0  0 0 0  0 0 0  0 1 0
```

$IFIX

```
some^fixed := $IFIX(-1, 3);
```

assigns the value -0.001F to "some^fixed".

$LFIX

```
some^fixed := $LFIX(-1, 3);
```

assigns the value 65.535F to "some^fixed".

$DFIX

```
some^fixed := DFIX(some^double, 3);
```

converts the value of "some^double" to its 64-bit integer equivalent then assigns the result to "some^fixed".

$FIXI

```
some^int := $FIXI( 234.05F );
```

assigns the value 23405 to "some^int".

Note that the Overflow indicator is set if the value of the <fixed expression> is less than -32768 or greater than 32767.

$FIXL

```
some^int := $FIXL(some^fixed);
```

assigns the least significant word of "some^fixed" too "some^int".

Note that the Overflow indicator is set if the value of the <fixed expression> is greater than 65535.

$FIXD

```
some^double := $FIXD(some^fixed);
```

assigns the two least significant words of "some^fixed" to "some^double".

Note that the Overflow indicator is set if the value of the <fixed expression> is less than -2,147,483,648 or greater than 2,147,483,647.

The character test functions are used to determine if a character falls into the range of ASCII alphabetical characters, ASCII numerical characters, or ASCII special characters. If the tested character passes the test, a true state (i.e., value of -1) is returned; if the test fails, a false state (i.e., value of 0) is returned.

The character test functions are typically used in conditional expressions to make decisions about the flow of program execution.

The general form for testing a character is:

```
$ALPHA ( <expression> )
------ - ------------ -
   tests <expression> for an ASCII alphabetical character:

      <expression> >= "A" AND <expression> <= "Z" OR
      <expression> >= "a" AND <expression> <= "z"

$NUMERIC ( <expression> )
-------- - ------------ -
   tests <expression> for an ASCII numerical character:

      <expression> >= "0" AND <expression> <= "9"

$SPECIAL ( <expression> )
-------- - ------------ -
   tests <expression> for an ASCII special character:

      <expression> <> alphabetical and <expression> <> numerical

where

   <expression> is a 16-bit value; the character tests check only
   <expression>.<8:15>, bits 0 through 7 are ignored

   the character tests also provide a condition code setting that
   indicates the class of character tested (if the state of the
   condition code is to be checked, it must be checked before an
   arithmetic operation is performed or an assignment is made to
   a variable)

      <   indicates numerical was found
      =   indicates alphabetical was found
      >   indicates special was found

example

   IF $ALPHA(some^char) THEN ...;
```

Some examples using the following declarations:

```
INT index;                          ! data declarations.
INT int^array[0:10];                !
STRING string^array[0:72];          !
```

$ALPHA

```
IF $ALPHA( string^array[index] ) THEN ... ;
```

test "string^array[index]" for being an ASCII alphabetical character.

$NUMERIC

```
DO index := index + 1 UNTIL NOT $NUMERIC( int^array[ index ] );
```

increments "index" until the right half of an element in "int^array" is found to be other than numerical.

$SPECIAL

```
WHILE $SPECIAL(string^array[index]) DO index := index - 1;
IF = THEN.......;
```

decrements index while ASCII special characters are encountered, then checks the condition code indicator for alphabetical character.

These two functions provide the signed minimum or maximum of two expressions.

The general form is:

```
$MIN ( <arithmetic expression> , <arithmetic expression> )
---- - ---------------------- - ---------------------- -
   returns the minimum of two signed integer, double word
   integer, or fixed expressions

$MAX ( <arithmetic expression> , <arithmetic expression> )
---- - ---------------------- - ---------------------- -
   returns the maximum of two signed integer, double word
   integer, or fixed expressions

$LMIN ( <arithmetic expression> , <arithmetic expression> )
----- - ---------------------- - ---------------------- -
   returns the minimum of two unsigned integer expressions

$LMAX ( <arithmetic expression> , <arithmetic expression> )
----- - ---------------------- - ---------------------- -
   returns the maximum of two unsigned integer expressions

example

   vary := $MIN( integer1, integer2);
```

Some examples using the following declarations:

```
INT smaller := 1,larger := 10,vary; ! data declaration.

PROC int^proc INT;

BEGIN
  RETURN smaller + larger;
END;
```

$MIN

```
vary := $MIN(smaller, larger);
```

assigns lesser of "smaller" and "larger" to "vary" (ie, 1).

```
WHILE  $MIN(larger, smaller) DO ...;
```

determines the lesser of "larger" and "smaller" then checks for the lesser value being nonzero

$MAX

```
IF $MAX(larger, int^proc * 2) > vary THEN.... ;
```

determines the greater of "larger" and "int^proc" times 2 then checks for greater value being greater than "vary".

Carry and Overflow Test Functions

These two tests check the states of the Carry and Overflow indicators in the machine. If the the tested indicator is on, a true state (i.e., value of -1) is returned; if the tested indicator is off, a false state (i.e., value of 0) is returned.

The Carry and Overflow tests are typically used in conditional expressions to make decisions concerning the flow of program execution.

The form of these tests are:

```
$CARRY
------
   test for a Carry condition

$OVERFLOW
---------
   tests for an Overflow condition

examples

   IF $CARRY THEN .... ;
   IF NOT $OVERFLOW THEN ... ;
```

The scale function is used to change the position of the implied decimal point of an expression. The point function returns the <fpoint> value associated with a fixed expression.

The form of the fixed point scale and point functions are:

$SCALE (<fixed expression> , <scale>)

moves the position of the implied decimal point by adjusting the internal representation of the expression (i.e., multiplying or dividing by <scale> power of ten)

Precision may be lost if the operand is scaled down

The Overflow indicator is set if the result of the scale exceeds the range of a 64-bit integer

where

<fixed expression> is an arithmetic expression giving a FIXED result

<scale> is the number of positions that the implied decimal point is to be moved to the left (<scale> > 0) or to the right (<scale> <= 0) of the least significant digit. <scale> is represented by an integer constant, the range of which is -19 to + 19

$POINT (<fixed expression>)

returns the <fpoint> value, in integer form, associated with a fixed expression.

Note: The compiler does not emit any instructions when evaluating the <fixed expression>. Therefore, <fixed expression> cannot be used invoke a function procedure or assign a value to a variable

examples

```
result := $SCALE(a, 3) / b;

point := $SCALE(a, $POINT( b )) / b;
```

Some examples using the following declarations:

```
INT point;                          ! data declarations.
FIXED(3) result, a, b;              !
```

$SCALE

```
result := $SCALE(a, 3) / b;
```

scales the value of "a" by +3 so that "a" is treated as a FIXED(6) value. The result of the divide operation is then a FIXED(3) value.

$POINT

```
result := $SCALE(a, $POINT( b )) / b;
```

This illustrates how to automatically retain precision when performing fixed point division. The operation is identical to the preceding example except that the $POINT function is used to determine the <scale> value to $SCALE. The value returned from $POINT is 3 (the <fpoint> value of <b>); <a> is scaled by that factor.

The compiler commands are used to control listing features, declare sections in source files, specify source files (and sections in source files) for compilation, specify the number of data pages desired in the object program, and to selectively compile portions of source files.

The general form of the compiler commands is:

```
?  <compiler command>
-  ------------------

where

   ? in column one of a source file line designates a compiler
   command line.  The compiler command line has one of the
   following forms:

      <command option> [ , <command option> ] ...

      <toggle command>

      <source command>

      Note that multiple <command options> can be specified per
      line.  If the line includes <source> or <toggle> commands, the
      command must be last in the line.  Therefore, <source> and
      <toggle> commands cannot appear in the same line

the <command options> consist of

   the page option:

      PAGE [ "<heading string>" ]

   the listing options ("->" indicates the default setting):

   -> LIST                NOLIST

   -> MAP                 NOMAP

   -> LMAP [ * ]          NOLMAP

   -> CODE                NOCODE

      ICODE           -> NOICODE

      INNERLIST       -> NOINNERLIST

      ABSLIST         -> NOABSLIST

                                                              -->
```

```
    -> WARN                NOWARN

    -> SUPPRESS            NOSUPPRESS

    the errors option:

       ERRORS [=] <max errors>

    the section option:

       SECTION <section name>

    the datapages option:

       DATAPAGES [=] <number of pages>

    the pep option:

       PEP [=] <pep table size>

    the fixed point rounding control option:

       ROUND           -> NOROUND

    the assertion control option:

      ASSERTION [=] <assertion level> , <procedure name>

  the <source command> is

    SOURCE <file name> [ ( <section name> , ... ) ]

  the <toggle commands> are

    SETTOG [ 1,2,..., n ]

    RESETTOG [ 1,2,..., n ]

    IF <toggle no.>

    IFNOT <toggle no.>

    ENDIF <toggle no.>
```

COMMAND OPTIONS

Page Command Option

PAGE [" <heading string> "]

The PAGE command option ejects the current page of the <list file>, prints the optional <heading string>, and skips two more lines. The <heading string> must be enclosed in quotes (which are deleted). PAGE is enabled only if the LIST option is on. If a <heading string> is specified, it replaces any preceding <heading string>.

Example:

?PAGE

causes a skip to the next top of page. The previous <heading string> (if any) is retained

?PAGE "FORWARD DECLARATIONS"

causes a skip to the next top of page. The string "FORWARD DECLARATIONS" is printed on that and subsequent page headings (until changed by another PAGE command)

Note: The PAGE command option is ignored if the <list> file is not a line printer. The first ?PAGE option in a source program does not cause a page eject. Rather it is used to specify a heading string.

Listing Command Options

Prefixing the following listing command options with "NO" disables that option:

LIST — Transmits each source image to <list file>. LIST also enables the other listing options.

MAP — If LIST is specified, a table of sublocal identifiers is printed following each subprocedure, a table of local identifiers is printed following each procedure, and a table of global identifiers is printed following the last procedure in the source program.

LMAP [*] — If LIST is specified, a table of procedure base addresses, entry point addresses, and limit addresses is printed following the last procedure in the source program.

The table is printed in alphabetical order of procedure names. If the form "LMAP*" is specified, the table is also printed in ascending order of procedure base addresses.

CODE — If LIST is specified, the instruction codes, in octal, are listed following each procedure.

ICODE — If LIST is specified, the mnemonics representing the instruction codes are printed following each procedure.

INNERLIST — If LIST is specified, the mnemonics representing the instruction codes are printed after each statement is compiled. Additionally, the compiler's RP (register stack pointer) setting is indicated.

INNERLIST is useful, when using the compiler interactively, for determining the specific machine instructions generated for a particular statement (i.e., to see how the compiler works).

ABSLIST — If LIST is specified, the instruction location printout will be relative to the base of the code area - C[0]. (Normally, the instruction locations are given relative to the base of a procedure.) If ABSLIST is to be used, the compiler must know the size of the PEP (procedure entry point) table before any actual procedure body is encountered in the source program. This can be accomplished in either of two ways: 1) by including a "PEP" command option at the beginning of the source program or 2) by having a FORWARD declaration for each internal procedure. These FORWARD declarations must precede any procedure having an actual body.

WARN — If WARN is specified, compiler warning messages are listed (regardless of the setting of LIST). If NOWARN is specified, compiler warning messages are suppressed.

SUPPRESS — If SUPPRESS is specified, all compiler listing output except error messages and the compiler's trailer message are suppressed. Use of this command overrides the state of the LIST option.

Examples:

```
?NOLIST
?LIST, LMAP*,NOCODE,ICODE
?INNERLIST
?SUPPRESS
```

Errors Command Option

ERRORS [=] <max errors>

The ERRORS command option limits the maximum number of errors allowed during a compilation to <max errors>. If this limit is exceeded, the compilation terminates. <max errors> is an integer in the range of {0:32767}. If omitted, the compilation continues regardless of the number of errors encountered.

Section Command Option

SECTION <section name>

The SECTION command option (which is used in conjunction with the SOURCE command) gives a name to a portion of a source file. A <section name> applies to all source text subsequent to the SECTION command or until another <section name> is given.

Note: A <section name> has the same characteristics as a T/TAL identifier. That is, it is composed of one to 31 contiguous letters, numbers, or circumflex symbols.

An example. Suppose a programmer has written a source library of application procedures. It might be desirable to give a <section name> to each procedure so that individual procedures could be selected for compilation using a SOURCE statement:

```
?SECTION sort^on^key
PROC sort^on^key(key1, key2, key3, length);
   INT .key1, .key2, .key3, length;
  BEGIN
    .
    .
  END;
?SECTION next^procedure
```

Then invoking a SOURCE command (with a file name of "appllib") as follows:

```
?SOURCE appllib (sort^on^key)
```

this compiles the procedure "sort^on^key".

Note: Although, in this example the name of the procedure was used as the section name, any name could have been used.

Datapages Command Option

DATAPAGES [=] <number of pages>

The DATAPAGES command option is used to override the number of data pages that the compiler assigns to the object program. The compiler, by default, assigns enough data pages for the global data area plus one extra page for local storage allocation.

If the number specified by DATAPAGES is not sufficient, the compiler assigns the default value.

Example:

```
?DATAPAGES = 64
```

Note: The number of data pages can be also be increased from the compiler assigned setting at run time. This is done via the MEM parameter of the RUN command or the <memory pages> parameter of the NEWPROCESS procedure.

Pep Command Option

```
PEP [=] <pep table size>
```

The PEP command option tells the compiler the anticipated size, in words, of the program's Procedure Entry Point Table. This command option is intended to be used when the ABSLIST listing option is specified so that the compiler will know how much space is to be allocated for the PEP. (This permits the compiler to list code relative addresses for instruction locations). The <pep table size> must be large enough to contain the PEP and can be specified as a larger value if desired.

Example:

```
?PEP 60
```

Fixed Point Rounding Control Command Option

```
ROUND
NOROUND                                  ! default setting.
```

These command options are used to determine if rounding should take place when a fixed value with one <fpoint> value is assigned to a fixed variable having a smaller <fpoint> value.

If NOROUND is specified, the value of a fixed operand that is assigned to a fixed variable is scaled, if necessary, to match the <fpoint> value of the fixed variable. If the <fpoint> value of the operand is greater than that of the variable, then the operand is scaled down and precision of the operand is lost.

If ROUND is specified, the value of a fixed operand is also scaled, if necessary, to match that of the assignment variable. If the <fpoint> value of the operand is greater than that of the variable, then the operand is first scaled, if necessary, so that its <fpoint> value is one greater than the variable. The scaled operand is then rounded as follows:

```
(IF operand < 0 THEN operand - 5 ELSE operand + 5) / 10
```

That is, if the operand is negative, 5 is subtracted; if positive, 5 is added. Then an integer divide by ten is executed to round the operand and scale it down by a factor of one. Therefore, if the absolute value of the least significant digit of the operand, after initial scaling, is 5 or greater, one is added to the absolute value of the final least significant digit.

Assertion Command Option

```
ASSERTION [=] <assertion level> , <procedure name>
```

The ASSERTION command option is a program debugging aid that is used in conjunction with an ASSERT statement. An ASSERT statement has the form:

```
ASSERT <assertion level> : <conditional expression>
```

The procedure specified by <procedure name> is invoked if the <conditional expression> is true and the level set by the ASSERTION command option is lower than or equal to the level specified in an ASSERT statement.

For example. While initially debugging a program, a call to the operating system DEBUG procedure is made if a certain unexpected condition is encountered. Therefore, the <assertion level> is set to five and the DEBUG procedure is specified:

```
?ASSERTION = 5, DEBUG
```

Then at some critical point in the program, an ASSERT statement is used:

```
      .
   SCAN array WHILE " " -> @pointer;
ASSERT 10: $CARRY
      .
```

The compiler generates the instruction codes necessary to check the carry indicator and call the DEBUG procedure. The result being that the DEBUG procedure is called if the carry indicator is set because of the SCAN (i.e., a condition that was not expected to occur).

Later on, when the program is fully debugged, the <assertion level> is raised to 20

```
?ASSERTION = 20, DEBUG
```

and the code necessary to check the carry indicator and call the DEBUG procedure is not generated by the compiler.

SOURCE COMMAND

```
?SOURCE <file name> [ ( <section name> , ... ) ]
```

The SOURCE command specifies a file (and optionally a section) from which source statements are to be compiled. <file name> is of the form

```
[$<volume name>.][<subvol name>.]<disc file name>
```

The current default volume and subvol names will be substituted for the corresponding omitted portions of <file name>.

The source file is processed until an end-of-file indication is encountered, at which point the compiler begins reading at the line following the SOURCE command. The maximum number of source files open at any given time (i.e., nested SOURCE commands) is four.

When the SOURCE command is included in a line with other compiler commands, the SOURCE commands must be the last commands in the line.

Example:

```
?SOURCE $SYSTEM.SYSTEM.EXTDECS
```

Section names can be continued on subsequent source lines. Column one of each continuation line must contain a question mark "?".

TOGGLE COMMANDS

The toggle commands are used to selectively compile portions of source files. A flag, called a toggle, is set or reset somewhere near the beginning of the source program. Commands are then entered at other points in the source program that test the toggles to determine if subsequent text should be compiled.

When the toggle commands are included in a line with other compiler commands, the toggle commands must be last in the line.

There are fifteen separate toggles available; all are initially reset. Toggles are set using:

```
?SETTOG [ 1, 2, ..., n ]
```

Sets the specified toggles. Omitting the toggle numbers, sets all toggles.

Toggles are reset using:

```
?RESETTOG [ 1, 2, ..., n ]
```

Clears the specified toggles. Omitting the toggle numbers, clears all toggles.

Toggles are tested using:

```
?IF <toggle no.>
```

ignores the subsequent text unless <toggle no.> is set.

```
?IFNOT <toggle no.>
```

ignores the subsequent text unless <toggle no.> is not set.

In any case, the compiler resumes compiling when

?ENDIF <toggle no.>

is encountered

Example:

```
?RESETTOG 1
  .
?IF 1
  .
 PROC some^proc;
   BEGIN
     .
     .
   END;

?ENDIF 1
```

The compiler skips over the source text between "IF 1" and "ENDIF 1"

The T/TAL compiler program resides in a file designated

$SYSTEM.SYSTEM.TAL

Normally, it is run through use of the Command Interpreter program. The command to run TAL is:

```
TAL [ / [ IN <source file> ] [, OUT [ <list file> ] ] / ]
---
    [ <object file> ] [ ; <compiler command> , ... ]
```

where

IN <source file>

specifies disc file, non-disc device, or process where TAL reads source language statements and compiler commands. TAL reads 132-byte records from <source file> until the end-of-file is encountered.

If this option is omitted, the <command file> of the Command Interpreter is used (if the Command Interpreter is being used interactively, which is the usual case, this will be the home terminal)

OUT <list file>

specifies a non-disc device, process, or existing disc file where TAL directs its listing output. If the <list file> is an unstructured disc file, each <list file> record is 132 characters (partial lines are blank filled through column 132).

If this option is omitted, the <list file> of the Command Interpreter is used (if the Command Interpreter is being used interactively, which is the usual case, this will be the home terminal)

<object file>

is a disc file, not necessarily existing, where the compiler puts the ready-to-run object program. It is of the form:

```
[$<volume name>.][<subvol name>.]<disc file name>
```

If omitted, a file designated

```
$<default volume>.<default subvol>.OBJECT
```

-->

is created. If a file exists with the same name as <object file> (or OBJECT, if applicable), the compiler purges the old file

<compiler command>

is one of

```
[NO]LIST
[NO]MAP
[NO]LMAP[*]
[NO]CODE
[NO]ICODE
[NO]INNERLIST
[NO]ABSLIST
[NO]WARN
[NO]SUPPRESS
[NO]ROUND
SETTOG [ 1,2,..., n ]
RESETTOG [ 1,2,..., n ]
PEP [ = ] <pep size>
ERRORS [ = ] <max errors>
```

See section 2.19 for an explanation of these commands

example

```
TAL/IN mysource, OUT $LP/myobject
```

DISC FILE SPACE USED BY THE COMPILER

During the course of a compilation, T/TAL makes use of five temporary disc files. Their sizes and where they reside are:

file	extent size	volume
1	5 * 2048 bytes	default volume
2	4 * 2048 bytes	default volume
3	4 * 2048 bytes	default volume
4	1 * 2048 bytes	default volume
5	depends on size of object program	volume specified in <object file>

The T/TAL cross reference program provides an alphabetical listing of all identifiers in a source program. The class of each identifier is given, as well as the procedure or subprocedure where declared (if applicable). Additionally, for each occurrence of an identifier in a source program, a number identifying the source file is given as well as the line number within the source file.

The cross reference program reads standard T/TAL source programs. The SOURCE, SECTION, and Toggle compiler commands are processed in the same manner as processed by the compiler. Other commands are ignored.

The T/TAL cross reference program resides in a file designated

```
$SYSTEM.SYSTEM.XREF
```

Normally, it is run through use of the Command Interpreter program. The command to run the cross reference program is:

```
XREF [ / [ IN <source file> ] [, OUT <list file> ] / ]
----

where

   IN <source file>

      specifies disc file, non-disc device, or process where XREF
      reads source language statements and compiler commands.
      XREF reads 132-byte records from <source file> until the
      end-of-file is encountered.

      If this option is omitted, the <list file> of the Command
      Interpreter is used (if the Command Interpreter is being
      used interactively, which is the usual case, this will be
      the home terminal)

   OUT <list file>

      specifies a non-disc device, process, or existing disc file
      where XREF directs its listing output. If the <list file>
      is an unstructured disc file, each <list file> record is 132
      characters (partial lines are blank filled).

      If this option is omitted, the <list file> of the Command
      Interpeter is used (if the Command Interpreter is being used
      interactively, which is the usual case, this will be the
      home terminal)

example

   XREF/IN mysource, OUT $LP/
```

The section describes the T/TAL Compiler listing format.

PAGE HEADING (?PAGE)

The Page Heading contains a page number, the name and compiler index number of the current source file being compiled, and an optional title if a ?PAGE command was specified:

```
PAGE 1     $SYSTEM.M013A00.NSTPSRCE  [1]
     ^                   ^            ^
     page number         file name    file number
```

COMPILER HEADING

The Compiler Heading gives the version number of the T/TAL compiler in use (and the data and time it was last updated), the current date and time, and the default listing options:

```
TAL - TANDEM COMPUTERS VERSION B00  ( 2/12/77 - 10 AM)
   SOURCE LANGUAGE: TAL - TARGET MACHINE TANDEM/16
DATE - TIME :  2/14/77 - 11:57:28
   OPTIONS: ON (LIST,CODE,MAP,WARN,LMAP) - OFF (ICODE,INNERLIST)
```

SEQUENCE NUMBERS AND SOURCE PROGRAM LINES

The sequence numbers are the EDIT line numbers of each source program line in the source file (in decimal).

```
 1.    000000 0  0  ?IF 1
 2.    000000 0  0  ?NOLIST
 3.    000000 0  0  ?ENDIF 1
 5.    000000 0  0
 6.    000000 0  0  ! this program is run with the following RUN c
 7.    000000 0  0  !
 8.    000000 0  0  ! :RUN xnonstop / IN <data base file>, OUT <te
 9.    000000 0  0  ! ---- -------- - --------------------
10.    000000 0  0  ! begin global declarations.
11.    000000 0  0
12.    000000 0  0  INT  backup^cpu,        ! this variable is not
13.    000000 0  0
14.    000000 0  0       io^busy := 0,      ! flag to determine if
15.    000000 0  0                          ! be restarted.  Set be
16.    000000 0  0                          ! cleared in <noerror>
17.    000000 0  0                          ! completes.
18.    000000 0  0
19.    000000 0  0      .recv^fname[0:11] := ["$RECEIVE", 8 * ["
 ^                  -->
 sequence numbers   source program lines
```

SECONDARY GLOBAL STORAGE

These numbers are a cumulative count of the amount of secondary global storage allocated (in octal). The count is relative to the first secondary global location (last primary global address + 1).

For example:

```
12.    000000   INT  backup^cpu,        ! this variable is not chec
13.    000000
14.    000000        io^busy := 0,      ! flag to determine if no-w
18.    000000
19.    000000       .recv^fname[0:11] := ["$RECEIVE", 8 * ["  "]],
20.    000014        recv^fnum,
21.    000014       .recv^buf[0:35],
22.    000060
       ^
       secondary storage allocation
```

The indirect array "recv^fname" is the first array allocated in secondary storage and therefore its storage is allocated starting in the initial secondary location (%000000). The "recv^fname" array occupies %14 words of secondary storage. Notice that the count does not increase when primary storage is allocated (e.g., for direct variable "recv^fnum"). The storage for indirect array "recv^buf" is allocated beginning at %14 of secondary storage and occupies %60 - %14 words of secondary storage (i.e, %44 words).

CODE ADDRESS

These numbers reflect the address (in octal) of the first instruction code that executes the T/TAL statement to its right. If the $ABSLIST listing option is not specified, the address is an offset from the base of the procedure (the first instruction in a procedure then has an offset of zero). If $ABSLIST is specified, the address is a C Relative address (i.e., offset from base of code area).

An example of code addresses produced without the $ABSLIST listing option:

```
69.    000000 0  0  PROC abort;
70.    000000 1  0
71.    000000 1  0    BEGIN
72.    000000 1  1      INT  p = 'L'[-2],
          .
          .
80.    000000 1  1      CALL MYTERM(buf);
81.    000014 1  1      CALL OPEN(buf,home^fnum);
       ^
       procedure relative address
```

The number %000014 indicates that the first instruction code associated with the statement on the right, "CALL OPEN..", is located in the %14th word of the procedure "abort".

To determine the actual C relative address of an instruction when the $ABSLIST option is not used, the base address of the procedure must be known. Procedures' base addresses are listed in the LMAP at the end of the compilation listing. In this example, the base of the procedure "abort" is in C relative location %000011. Therefore, the instruction is in C relative location %000025 (%11 (base) + %14 (offset from base)).

```
LMAP:

PEP    BASE    LIMIT   ENTRY  ATTRIBUTES  NAME

002    000011  000102  000011             ABORT
 .
 .
```

LEXICAL LEVEL

This number indicates the lexical level; 0 = global, 1 = procedure, 2 = subprocedure.

For example

```
 63.          0     ! end of global declarations.
 68.          0
 69.          0     PROC abort;
 70.          1
 71.          1       BEGIN
 72.          1         INT  p = 'L'[-2],
   .          .
   .          .
   .          .
 90.          1         CALL ABEND;
 91.          1       END;

269.          0     ! this is the transaction loop.
270.          0
271.          0     PROC main^loop;
272.          1
273.          1       BEGIN
   .          .
   .          .
279.          1         SUBPROC get^term^entry (error);
280.          2            INT .error;
281.          2           BEGIN
282.          2             INT reason;
   .          .
   .          .
   .          .
```

```
297.          2
298.          2          END; ! get^term^entry.

349.          1        WHILE 1 DO
350.          1          BEGIN
351.          1            CALL get^term^entry (error);
352.          1            IF error = 1 THEN RETURN; ! eof. signa
   .          .
   .          .
   .          .
385.          1          END; ! of infinite loop.
386.          1      END; ! of main^loop;
387.          0
              ^
              lexical level
```

BEGIN/END COUNTER

This number is the BEGIN/END counter. One is added each time a BEGIN is encountered, one is subtracted each time an END is encountered:

```
138.              0  PROC mycheckpoint;
139.              0
140.              0    BEGIN
141.              1      INT reason;
142.              1
143.              1      DO
144.              1        BEGIN
145.              2          CASE (reason := CHECKPOINT(stackbase,,
146.              2            BEGIN
147.              3                ! 0, good checkpoint.
   .              .
   .              .
152.              3              BEGIN
153.              4                CASE reason.<8:15> OF
154.              4                  BEGIN
155.              5                      ! 0, primary stopped.
   .              .
   .              .
161.              5                  END;
162.              4
   .              .
   .              .
168.              4              END;
169.              3                ! 3, parameter error.
171.              3            END;
172.              2        END
173.              1      UNTIL reason.<0:7> < 2; ! repeat checkpoin
174.              1    END; ! mycheckpoint.
                       ^
                       begin/end counter
```

MAP (?MAP)

The "MAP" listing option provides a map of identifiers. Three levels of maps are provided: sublocal (i.e., identifiers declared in a subprocedure), local (i.e., identifiers declared in a procedure), and global (i.e., identifiers declared globally). The form of a map is:

<identifier> <class> <type> <rel address> <address mode>

where

<class> specifies the class associated with the <identifier> (i.e., VARIABLE, PROC, SUBPROC, ENTRY, LABEL, DEFINE, LITERAL, etc.)

<type> is the declared data type of a VARIABLE. (i.e., INT, INT(32), STRING, or FIXED)

<rel address> is the relative address of a VARIABLE. In subprocedures addresses for both parameters and sublocal variables are addressed S - (verbally, S minus) relative. In procedures, parameters are addressed L - relative, local variables are addressed L +. All global variables are addressed G + relative

<address mode> indicates whether a VARIABLE is addressed directly or indirectly (i.e., through a pointer). Value parameters are indicated by DIRECT addressing; reference parameters are indicated by INDIRECT addressing.

An example map of local identifiers:

```
DB^EOF                  VARIABLE    INT     L+003          DIRECT
ERROR                   VARIABLE    INT     L+001          DIRECT
GET^TERM^ENTRY          SUBPROC
LIST^DB^ENTRY           SUBPROC
LIST^MODE               VARIABLE    INT     L+002          DIRECT
NEXT^DB^ENTRY           SUBPROC

^                       ^           ^       ^              ^
identifier              class       type    rel address    mode
```

CODES (?CODE)

The "CODE" listing option lists the instruction codes emitted for a procedure. The format of the code listing is:

```
<address>  <code> ... ! to eight per <codes> per <address>

where

   <address> is address (in octal) for adjacent code.  If the
   $ABSLIST listing option is not specified, <address> is an
   offset from the base of the procedure.  If $ABSLIST is
   specified, <address> is a C Relative address (i.e., offset
   from base of code area)

   <code> is the octal representation of the instruction code
   emitted by the compiler
```

An example instruction code listing:

```
00000     040001 007100 015460 040005 014404 040005 100000 024711
00010     027000 170040 000002 040001 006017 170037 070000 024755
00020     027000 040000 015440 040001 004440 044001 170037 070005
00030     100000 024722 027000 015025 100000 144031 040005 170031
00040     103777 143031 024722 002003 100034 024700 027000 012005
00050     170037 100001 024711 027000 010403 040001 004500 044001
00060     010402 100000 044005 125003
^         -->
address   codes
```

PROCEDURE MAP (?LMAP)

The "LMAP" listing option provides a map of all internal procedures in the program. The form of the map is:

```
PEP     BASE     LIMIT    ENTRY    ATTRIBUTES        NAME

<adr>   <adr>    <adr>    <adr>    [P|C|R|I|M|V]   <procedure name>
  .       .        .        .           .               .
  .       .        .        .           .               .
  .       .        .        .           .               .
```

where

PEP <adr> is the 'C' relative address, in octal, of the PEP entry for <procedure name>

BASE <adr> is the 'C' relative address, in octal, of the base (i.e., first word) of <procedure name>

LIMIT <adr> is the 'C' relative address, in octal, of the limit (i.e., last word) of <procedure name>

ENTRY <adr> is the 'C' relative address, in octal, of the entry point (i.e., first instruction) of <procedure name>

ATTRIBUTES are any attributes of <procedure name>. Where

- P = PRIV
- C = CALLABLE
- R = RESIDENT
- I = INTERRUPT
- M = MAIN
- V = VARIABLE

<procedure name> is the name of an internal procedure in the source program

An example procedure map is:

```
PEP     BASE     LIMIT    ENTRY   ATTRIBUTES   NAME

002     000011   000102   000011               ABORT
003     000103   000212   000103               CREATEBACKUP
004     000522   001000   000700               MAIN^LOOP
005     000213   000305   000213               MYCHECKPOINT
006     000427   000521   000427               NOERROR
007     001001   001235   001001     M         STARTUP
010     000306   000426   000306     V         WAIT
```

COMPLETION MESSAGE

The completion message gives the following information (all numeric values are given in decimal representation):

* The name of the object file created as a result of the compilation.
* The total number of error messages issued.
* The total number of warning messages issued.
* The total number of words of primary global storage needed.
* The total number of words of secondary global storage needed.
* The total number of words of code area needed.
* The minimum number of (virtual) memory data pages to be allocated to the program when it is run.
* The number of (virtual) memory code pages to be allocated to the program when it is run.
* The number of words that the compiler needed for its symbol table.
* The elapsed (or wall) time used to compile the source program.

An example completion message:

```
OBJECT FILE NAME IS $SYSTEM.M013A00.XNSTP
NO. ERRORS=0    ; NO. WARNINGS=0
PRIMARY GLOBAL STORAGE=18
SECONDARY GLOBAL STORAGE=208
CODE SIZE=661
DATA AREA SIZE=2  PAGES
CODE AREA SIZE=1  PAGES
MAXIMUM SYMBOL TABLE SIZE=887
NUMBER OF SOURCE LINES=421
ELAPSED TIME -   0: 1:25
```

Notes:

1. 'S' and 'L', when the object program is run, are initially set to

   <primary global storage> + <secondary global storage>

2. The number of (virtual) memory data pages can be increased when the program is run.

The following advanced features of T/TAL are discussed in this section:

* Base address equivalencing
* Procedures: advanced <attributes>
* Subprocedures: <attribute> variable
* Symbol for removing indirection
* Advanced statements
* Advanced standard functions
* Advanced compiler control commands

Base address equivalencing provides a method for assigning variables to locations relative to the four base addresses used in the Tandem 16. No storage is allocated for a variable equivalenced to a base address.

The form for base reference equivalencing is:

```
                        { 'G'  }
                        { 'L'  }
                        { 'S'  } [ "[" <word index> "]" ]
<type> [ . ] <name> = { 'SG' } [ [+|-] <word offset>  ] ;
```

where

<type> is { INT }
{ INT(32) }
{ STRING }
{ FIXED [(<fpoint>)] }

. is the indirection symbol. It presence means that the equivalenced variable is treated as a pointer variable. Its absence means that the equivalenced variable it treated as a simple variable

'G', 'L', 'S', and 'SG' are the address bases

where

'G' indicates global addressing - the variable is addressed relative to 'G'[0]

'L' stands for local addressing - the variable is addressed relative to 'L'[0]

'S' stands for top-of-stack addressing - the variable is addressed relative to 'S'[0]

'SG' stands for system global addressing - the variable is addressed relative to 'SG'[0]. This requires privileged mode

<word index> and <word offset> are integer constants and are equivalent

Example

```
INT l = 'L';
```

The location indicated by <word index> and <word offset> must be within the range of direct addressing for the particular base address:

for 'G' the range is [0:255]

for 'L' the range is [-31:127]

for 'S' the range is [-31:0]

for 'SG' the range is [0:63]

Some examples:

To access the upper 32K words of the data area, the following base register equivalencing could be used:

```
INT g = 'G';
```

base of the data area

Then using the identifier "g"

```
n := g[%100000];
```

accesses data area location 'G'[32768].

To equivalence a variable to the current location of the stack marker, the following is written:

```
INT mark = 'L';
```

Then using the identifier "mark"

```
n := mark;
```

accesses the "L" word in the current stack marker.

To equivalence a variable to the "E" part of the current stack marker, the following could be written:

```
INT e = 'L'[-1];
```

Base address equivalencing is used to access the system data area (if in privileged mode):

```
INT system = 'SG';
```

Then used as follows:

```
n := system[num];
```

accesses 'SG'[num] if in privileged mode. If not in privileged mode, 'G'[num] is accessed.

Procedures have a number of <attributes> not previously mentioned. That is:

* The ability to have a variable number of parameters passed at the time of the call.

* To execute in privileged mode but be callable by non-privileged procedures.

* To execute in privileged mode but be callable only from procedures already executing in privileged mode.

* To be designated an interrupt procedure and execute an IXIT instruction when returning to the caller rather than an EXIT instruction.

<attributes> are designated in the procedure heading. The complete form of the procedure heading is:

without parameters

```
[ <type> ] PROC <name> [ <attributes> ] ;
```

with parameters

```
[ <type> ] PROC <name> ( <formal parameter name> , ... )
   [ <attributes> ] ;
   <parameter specifications>
```

where

<type>, <name>, <formal parameter name>, and <parameter specifications> are described in "Procedure Declaration"

<attributes> are [MAIN [, RESIDENT [, VARIABLE
[, CALLABLE | PRIV [, INTERRUPT]]]]]

where

MAIN indicates that the procedure is the first one to execute when the program is run

RESIDENT indicates that the procedure's instruction codes are to be made main memory resident when the program is run

VARIABLE indicates that the procedure's <formal parameters> are to be considered optional and any or all parameters can be omitted at the time that the procedure is called

CALLABLE indicates that the procedure executes in privileged mode and is callable by procedures executing in non-privileged mode

PRIV indicates that the procedure executes in privileged mode and is callable only by other procedures executing in privileged mode

INTERRUPT indicates that the procedures executes an IXIT instruction when returning to the caller. This <attribute> is used only by the operating system interrupt handlers

<attribute> VARIABLE

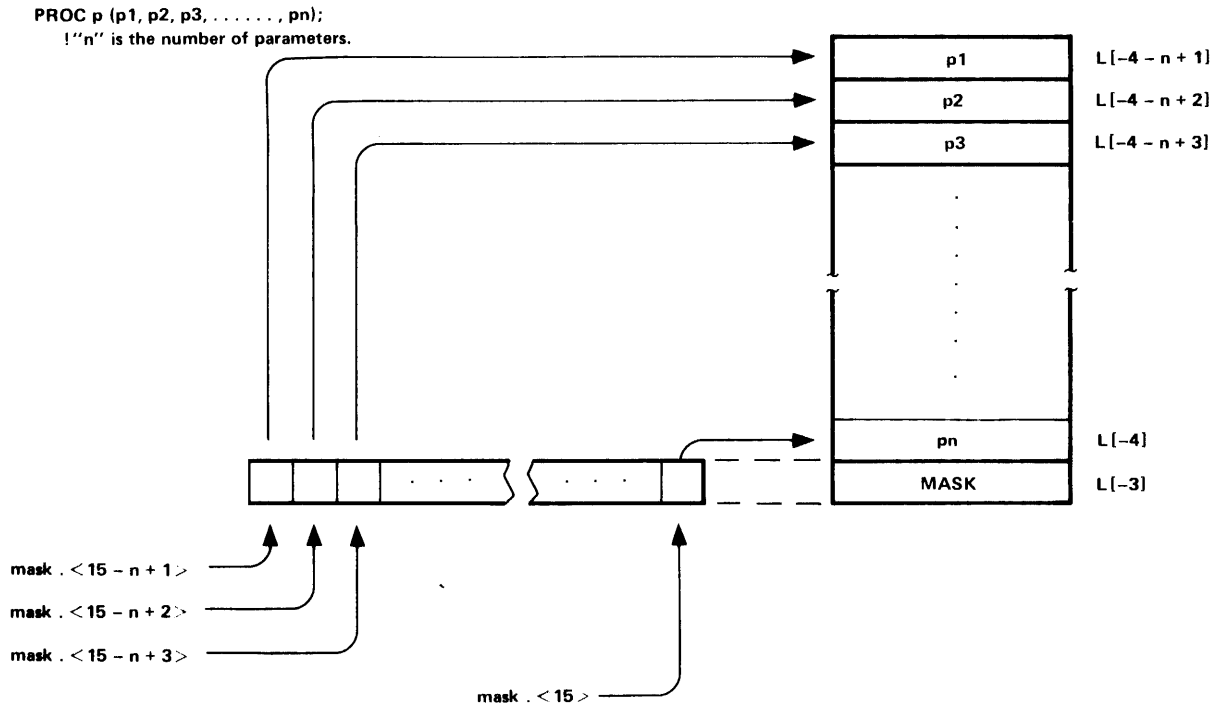
When a procedure is written with the <attribute> VARIABLE, the compiler considers ALL parameters to be optional. At the call to a procedure having this <attribute>, the compiler emits instruction codes to generate a parameter "mask". The "mask" is used to indicate the presence or absence of each parameter and is located in the parameter area just above the procedure's parameters.

The parameter "mask" consists of one word if the procedure has sixteen or less parameters or a doubleword if the procedure has more than sixteen parameters. If a one word mask, it is placed in 'L'[-3]; if a doubleword, it is placed in 'L'[-4:-3]. Each bit in the mask corresponds to one of the procedure's <formal parameters>. The mask bit is set to a "1" if the corresponding actual parameter is supplied at the time of the call and a "0" if the parameter is omitted. It is the responsibility of statements within a procedure with the <attribute> VARIABLE to check this mask.

The mask has the form:

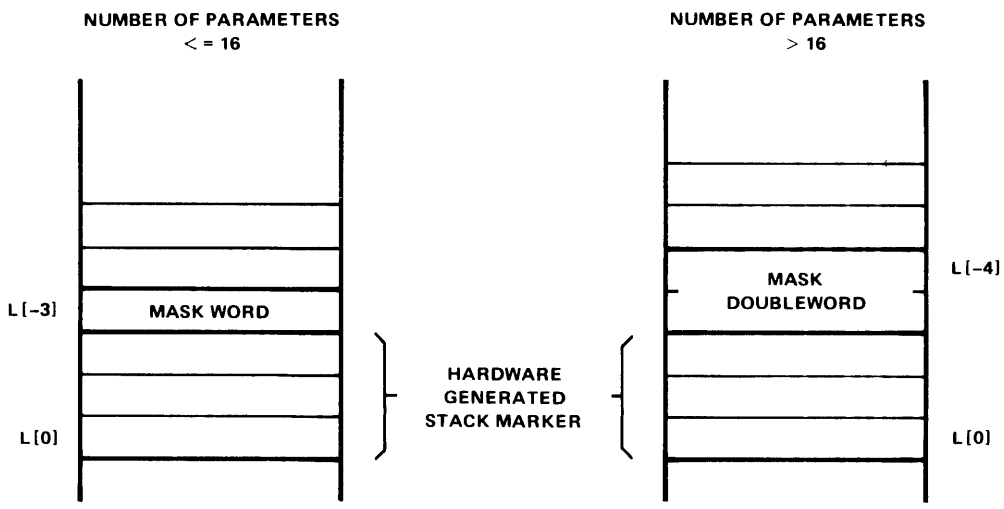
```
for VARIABLE procedure <p>, with left-to-right parameters -
<p1>, <p2>, <p3>, .. <pn> - that is:

   PROC p (p1, p2, p3, ... pn);

the corresponding parameter mask bits (for a one word mask) are

   for <p1>, <mask>.<15 - n + 1)
   for <p2>, <mask>.<15 - n + 2)
   for <p3>, <mask>.<15 - n + 3)
     .
     .
   for <pn>, <mask>.<15>
```

PARAMETER MASK FORMAT



The parameter mask resides beginning at 'L'[-3] if the procedure has 16 or less parameters and at 'L'[-4] if the procedure has from 17 to 27 parameters (27 is the maximum amount for variable procedures).

PARAMETER MASK LOCATION



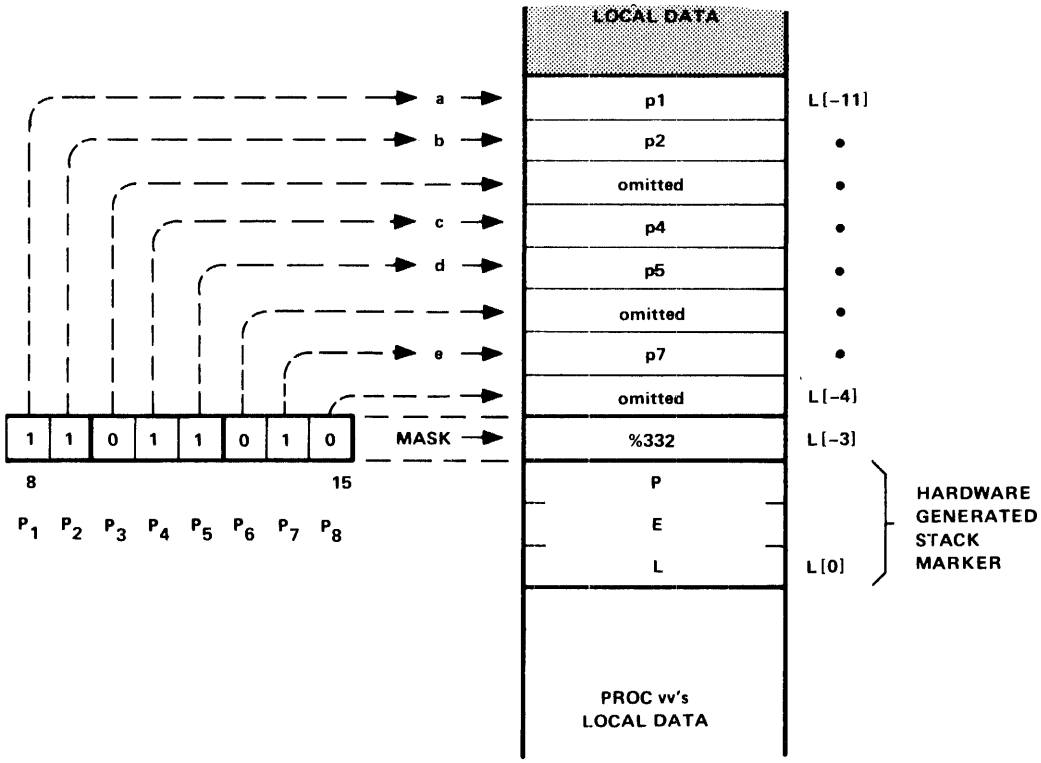
The following illustration depicts the parameter area for a call to a procedure having optional parameters:

EXAMPLE OF PARAMETER AREA FOR VARIABLE PROC

```
INT a,b,c,d,e;                          ! data declarations.

PROC vv(p1,p2,p3,p4,p5,p6,p7,p8) VARIABLE;
   INT p1,p2,p3,p4,p5,p6,p7,p8;

  BEGIN
    .
    .
  END;
```

The following procedure call is made:

```
CALL vv(a,b,,c,d,,e);
```



A standard function, $PARAM, is available for checking the presence or absence of a parameter. $PARAM is a <primary>.

The form of $PARAM is:

```
$PARAM ( <formal parameter> )
------ - ------------------ -

where

   $PARAM returns a "1" if the parameter is present and returns a
   "0" if the parameter is absent

   <formal parameter> is the name, as specified in the procedure
   (or subprocedure) heading, of the parameter whose presence is
   to be checked

example

   IF $PARAM(pl) THEN ..
```

An example using the following procedure declaration:

```
PROC var^proc (buffer, length, key) VARIABLE;
   INT .buffer, ! required.
        length, ! required.
        key;    ! optional.
```

The presence of an optional parameter is checked:

```
IF $PARAM( key ) THEN ...
```

   is true if the optional parameter "key" is present.

It is up to the programmer to ensure that required parameters are present. Required parameters are checked in the same manner as optional parameters. For example:

```
IF NOT $PARAM( buffer ) AND NOT $PARAM( length ) THEN ..
```

   this condition is true if both of the required parameters are absent.

<attribute> CALLABLE

The CALLABLE <attribute> is a nonprivileged program's only way to become privileged. Procedures declared with this attribute are callable by nonprivileged procedures.

An example. An application wants to access data in the operating system's data area. This can be done by non-system programs only if they are executing in privileged mode. This is accomplished as follows:

```
INT PROC read^system(address) CALLABLE;
   INT address;
  BEGIN
    INT sg = 'SG'; ! system global addressing mode

    RETURN sg[address];
  END
```

The procedure uses the 'SG' addressing mode which is possible only in privileged mode

<attribute> PRIV

Procedures declared with the PRIV attribute are callable only by procedures executing in privileged mode. The PRIV attribute is the operating system's mechanism for protecting against unauthorized calls to internal operating system procedures.

That is:

```
nonprivileged ----> CALLABLE ----> PRIV

(application)       (operating system)
```

<attribute> INTERRUPT

Procedures declared with the INTERRUPT attribute execute an IXIT (interrupt exit) instruction when returning to the caller. The INTERRUPT attribute is usable only by the operating system interrupt handlers.

Like a procedure, a subprocedure may have optional parameters by specifying the attribute VARIABLE. The <attribute> VARIABLE is declared in the subprocedure heading. The form of the <subprocedure heading> is:

without parameters

```
[ <type> ] SUBPROC <name> [ VARIABLE ] ;
```

with parameters

```
[ <type> ] SUBPROC <name> ( <formal parameter name> , ... )
   [ VARIABLE ] ;

   <parameter specifications>
```

where

<type>, <name>, <formal parameter name>, and <parameter specifications> are described in "Subprocedure Declaration"

VARIABLE is an <attribute> that indicates that the subprocedure's <formal parameters> are to be considered optional and any or all parameters can be omitted at the time that the subprocedure is called

When a subprocedure is written with the <attribute> VARIABLE, the compiler considers ALL parameters to be optional. At the call to a subprocedure having this <attribute>, the compiler emits instruction codes to generate a parameter "mask" of the same form as that for a call to a procedure having the <attribute> VARIABLE.

The parameter "mask", whether one or two words, is located in the memory stack just above where the right-hand parameter of the subprocedure is passed and just below where the caller's return address is stored.

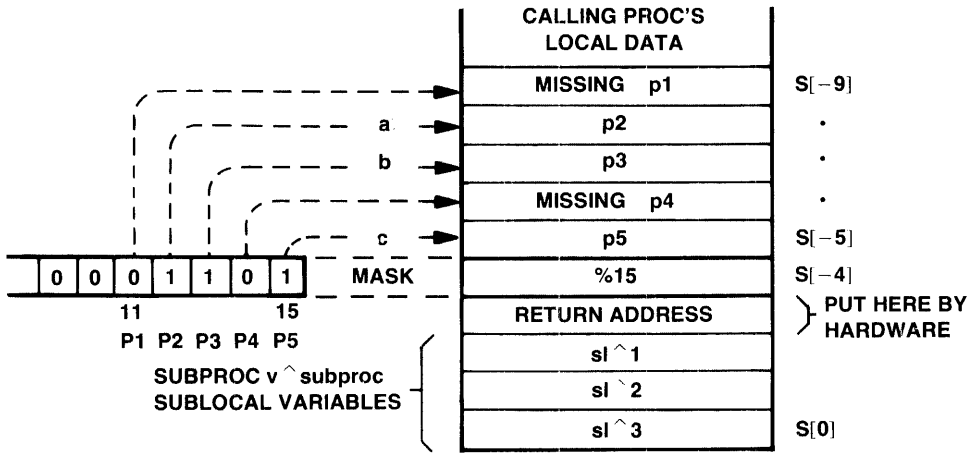
The following illustration depicts the parameter area for an call to a subprocedure having optional parameters:

EXAMPLE OF PARAMETER AREA FOR VARIABLE SUBPROC

```
INT a,b,c;

SUBPROC v^subproc(p1, p2, p3, p4, p5) VARIABLE;
   INT p1, p2, p3, p4, p5;

  BEGIN
    INT sl^1, sl^2, sl^3;
    .
    .
  END;
```

The following subprocedure call is made:

```
CALL v^subproc(,a,b,,c);
```



Note that the parameter "mask" is located two words before the first sublocal variable:

```
INT mask = sl^1 - 2;
```

This is the mask position in all cases for a one-word mask. If a subprocedure has more than 16 parameters, the mask consists of two words and is located starting three words before the first sublocal variable:

```
INT mask1 = sl^1 - 3,
    mask2 = mask1 + 1;
```

The standard function $PARAM (see "Procedures: Advanced <attributes>") can be used to check for the presence of parameters passed to a subprocedure with the attribute VARIABLE:

```
IF $PARAM( p1 ) THEN ...
```

The symbol for removing indirection can be used to obtain an address relative to the base of the code area ('C'[0]) of a label or subprocedure:

```
@<label> or @<subproc name>
```

For example, to obtain the address of the label

```
loop:
```

the following statement could be written

```
address := @loop;
```

The symbol for removing indirection can be used to obtain the procedure entry point (PEP) table number of a procedure:

```
@<proc name>
```

For example, to obtain the PEP number of the procedure

```
PROC p;
```

the following statement could be written

```
n := @p;
```

LABEL DECLARATION

Because identifiers for labels could duplicate globally declared identifiers, an ambiguity could arise when assigning the address of a label to a variable. Therefore the label declaration is provided.

The form of the label declaration is:

```
LABEL <label name> ;
----- ------------ -

where

   <label name> is an identifier assigned to the labelled
   statement

   more than one <label name> can be specified per declaration
   (separated by commas ",")

example

   LABEL loop;
```

An example showing the reason for the LABEL declaration. A variable is declared globally

```
INT loop;
```

Then in a procedure, the same identifier is used for a label:

```
PROC p;
  BEGIN
    .
    LABEL loop;
```

The address of the label "loop" is stored in the variable "n":

```
    n := @loop;
      .
      .
    loop:
      .
```

If "loop" had not been declared a label, the 'G'[0] relative address of the global variable "loop" would have been stored in "n".

The use of the following statements requires an advanced knowledge of the Tandem 16 hardware. These statements are:

CODE	permits the programmer to specify machine level instruction codes to be compiled into the object program
USE	allocates an index register permitting the programmer to optimize index register use
DROP	deallocates an index register
STACK	loads the contents of variables onto the register stack
STORE	stores register stack elements into variables
FOR	uses the BOX instruction to provide the loop if an index register has been allocated

The CODE statement provides the means for generation of specific instruction codes.

The general form of the CODE statement is:

```
CODE ( <instruction> ; ... )
---- - -------------       -
```

where

<instruction> is a Tandem 16 mnemonic representing the machine instruction to be executed, followed by one or more parameters if necessary

<instruction> has six forms, represented by six classes. The forms are:

class 1 - <mnemonic>

class 2 - <mnemonic> <variable>

class 3 - <mnemonic> <constant>

class 4 - <mnemonic> <index register>

class 5 - <mnemonic> <variable> [, <index register>]

class 6 - <mnemonic> <constant> [, <index register>]

where

<mnemonic> represents an instruction code as described in the Tandem 16 System Description and in this section

<variable> can be represented by

<identifier>

.<identifier>

@<identifier>

If <variable> is declared as an indirect variable and is specified without "@", the instruction emitted will be an indirect reference through <variable>

<index register> is an integer constant specifying an index register (i.e., [5:7]) or an identifier assigned to an index register in a USE statement. If omitted, no indexing is performed

Some examples showing the different classes of instructions:

```
CODE(ZERD; IADD);
```

These are class 1 instructions

```
CODE(LADR a; STOR .b);
```

These are class 2 instructions

```
CODE(LDI 21; ADDI -4);
```

These are class 3 instructions

```
CODE(STAR 7; STRP 2);
```

These are class 4 instructions

```
CODE(LDX a,7; LDB .stg, x);
```

These are class 5 instructions

```
CODE(LDXI -15,5);
```

This is a class 6 instruction

TANDEM 16 INSTRUCTION SET MNEMONICS

16-Bit Arithmetic (implicit top of Register Stack)

Integer (signed) Add A to B CODE (IADD)	! class 1
Logical (unsigned) Add A to B CODE (LADD)	! class 1
Integer (signed) Subtract A from B CODE (ISUB)	! class 1
Logical (unsigned) Subtract A from B CODE (LSUB)	! class 1
Integer (signed) Multiply A times B CODE (IMPY)	! class 1
Logical (unsigned) Multiply A times B CODE (LMPY)	! class 1
Integer (signed) Divide B by A CODE (IDIV)	! class 1
Logical (unsigned) Divide CB by A CODE (LDIV)	! class 1
Integer (signed) Negate A CODE (INEG)	! class 1

-->

```
|                                                                    |
|      Logical (unsigned) Negate A                                   |
|         CODE ( LNEG )                                ! class 1     |
|      Integer (signed) Compare A with B                             |
|         CODE ( ICMP )                                ! class 1     |
|      Logical (unsigned) Compare A with B                           |
|         CODE ( LCMP )                                ! class 1     |
|      Compare Immediate Operand with A                              |
|         CODE ( CMPI <immediate operand> )            ! class 3     |
|      Add Immediate Operand to A                                    |
|         CODE ( ADDI <immediate operand> )            ! class 3     |
|      Logical (unsigned) Add Immediate Operand to A                 |
|         CODE ( LADI <immediate operand> )            ! class 3     |
|                                                                    |
|  32-Bit Signed Arithmetic (implicit top of Register Stack)         |
|  ---------------------------------------------------------        |
|                                                                    |
|      Double Add DC to BA                                           |
|         CODE ( DADD )                                ! class 1     |
|      Double Subtract BA from DC                                    |
|         CODE ( DSUB )                                ! class 1     |
|      Double Negate BA                                              |
|         CODE ( DNEG )                                ! class 1     |
|      Double Compare BA with DC                                     |
|         CODE ( DCMP )                                ! class 1     |
|      Double Test BA                                                |
|         CODE ( DTST )                                ! class 1     |
|      Minus One Double                                              |
|         CODE ( MOND )                                ! class 1     |
|      Put Zero Double into BA                                       |
|         CODE ( ZERD )                                ! class 1     |
|      One Double                                                    |
|         CODE ( ONED )                                ! class 1     |
|                                                                    |
|  16-Bit Signed Arithmetic (explicit Register Stack Element)        |
|  ---------------------------------------------------------        |
|                                                                    |
|      Add Register to A                                             |
|         CODE ( ADRA  <register number> )             ! class 4     |
|      Subtract Register from A                                      |
|         CODE ( SBRA <register> )                     ! class 4     |
|      Add A to a Register                                           |
|         CODE ( ADAR <register> )                     ! class 4     |
|      Subtract A from a Register                                    |
|         CODE ( SBAR <register> )                     ! class 4     |
|      Add Immediate Operand to an Index Register                    |
|         CODE ( ADXI <immediate operand> , <index register> )       |
|                                                      ! class 6     |
|                                                                    |
|                                                                    |
|                                                                    |
|                                                                --> |
```

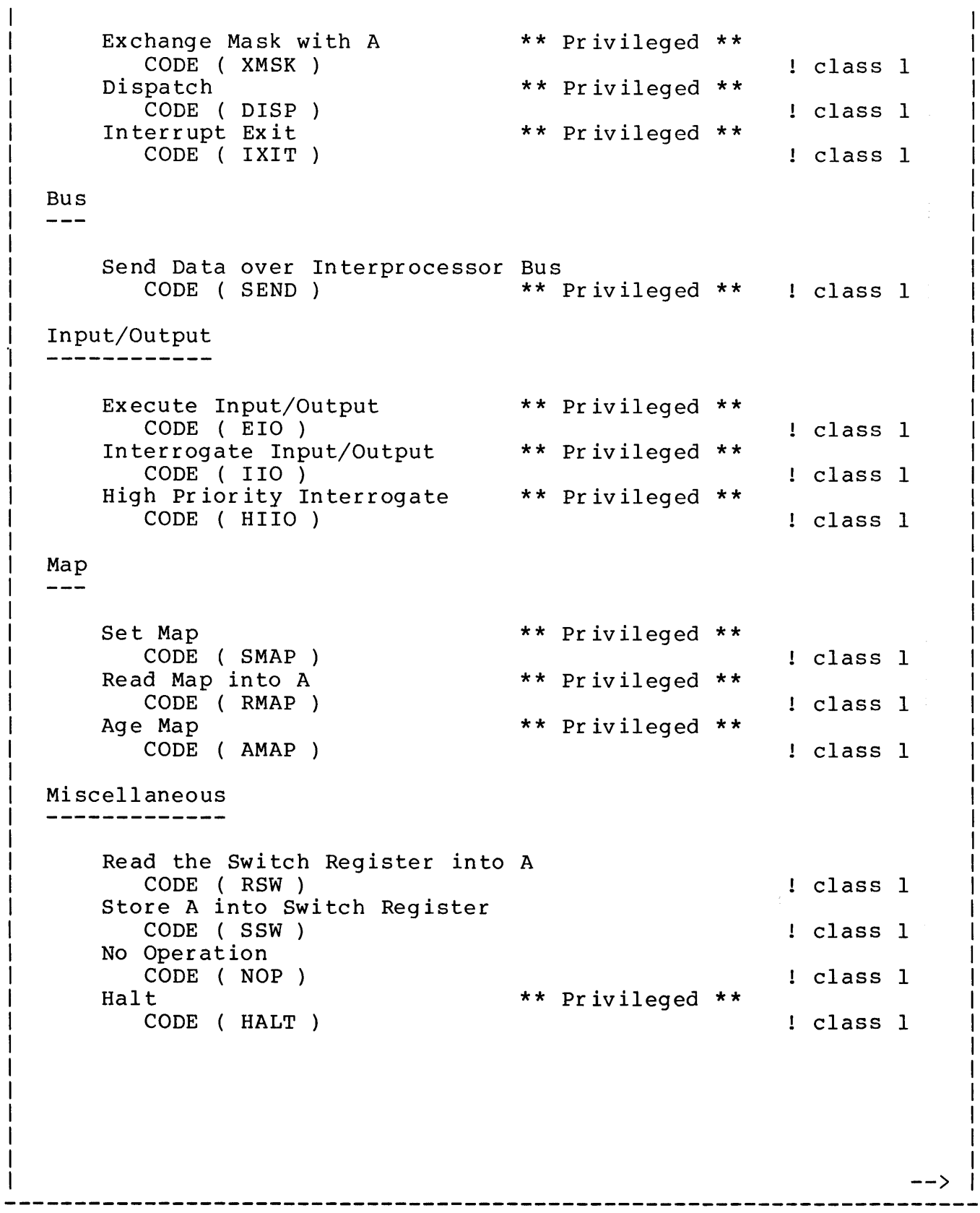
```
Register Stack Manipulation
---------------------------

    Exchange A and B
       CODE ( EXCH )                                    ! class 1
    Double Exchange BA with DC
       CODE ( DXCH )                                    ! class 1
    Double Duplicate BA in DC
       CODE ( DDUP )                                    ! class 1
    Store A in a Register
       CODE ( STAR <register> )                         ! class 4
    Non-destructive Store A into a Register
       CODE ( NSAR  <register> )                        ! class 4
    Load A from a Register
       CODE ( LDRA <register> )                         ! class 4
    Load Immediate Operand into A
       CODE ( LDI <immediate operand> )                 ! class 3
    Load Index Register with Immediate Operand
       CODE ( LDXI <immediate operand> , <index register> )
                                                        ! class 6
    Load Left Immediate Operand
       CODE ( LDLI <immediate operand> )                ! class 3

Boolean Operations
------------------

    Logical (unsigned) AND A with B
       CODE ( LAND )                                    ! class 1
    Logical (unsigned) OR A with B
       CODE ( LOR )                                     ! class 1
    Logical (unsigned) Exclusive OR A with B
       CODE ( XOR )                                     ! class 1
    One's Complement A
       CODE ( NOT )                                     ! class 1
    OR Right Immediate Operand with A
       CODE ( ORRI <immediate operand> )                ! class 3
    OR Left Immediate Operand with A
       CODE ( ORLI <immediate operand> )                ! class 3
    AND Right Immediate Operand to A
       CODE ( ANRI <immediate operand> )                ! class 3
    AND Left Immediate Operand with A
       CODE ( ANLI <immediate operand> )                ! class 3

Bit Shift and Deposit
---------------------

    Deposit Field in A
       CODE ( DPF )                                     ! class 1
    Logical (unsigned) Left Shift
       CODE ( LLS <shift count> )                       ! class 3
                                                                -->
```

```
|                                                                    |
|      Double Logical (unsigned) Left Shift                          |
|         CODE ( DLLS <shift count> )                ! class 3       |
|      Logical (unsigned) Right Shift                                |
|         CODE ( LRS <shift count> )                 ! class 3       |
|      Double Logical (unsigned) Right Shift                         |
|         CODE ( DLRS <shift count> )                ! class 3       |
|      Arithmetic (signed) Left Shift                                |
|         CODE ( ALS <shift count> )                 ! class 3       |
|      Double Arithmetic (signed) Left Shift                         |
|         CODE ( DALS <shift count> )                ! class 3       |
|      Arithmetic (signed) Right Shift                               |
|         CODE ( ARS <shift count> )                 ! class 3       |
|      Double Arithmetic (signed) Right Shift                        |
|         CODE ( DARS <shift count> )                ! class 3       |
|                                                                    |
|  Byte Test                                                         |
|  ----------                                                        |
|                                                                    |
|      Byte Test A                                                   |
|         CODE ( BTST )                              ! class 1       |
|                                                                    |
|  Memory Stack <--> Register Stack                                  |
|  --------------------------------                                  |
|                                                                    |
|      Load Word from Program (Code) Area into A                     |
|         CODE ( LWP <variable> , <index register> )  ! class 5      |
|      Load Byte from Program (Code) Area into A                     |
|         CODE ( LBP <label> , <index register> )     ! class 5      |
|      Load Index Register from Memory Stack                         |
|         CODE ( LDX <variable> , <index register> )  ! class 5      |
|         CODE ( NSTO <variable> , <index register> ) ! class 5      |
|      Load A from Memory Stack                                      |
|         CODE ( LOAD <variable> , <index register> ) ! class 5      |
|      Store A into Memory Stack                                     |
|         CODE ( STOR <variable> , <index register> ) ! class 5      |
|      Load A with Byte from Memory Stack                            |
|         CODE ( LDB <variable> , <index register> )  ! class 5      |
|      Store Byte from A.<8:15> to Memory Stack                      |
|         CODE ( STB <variable> , <index register> )  ! class 5      |
|      Load Double from Memory Stack into BA                         |
|         CODE ( LDD <variable> , <index register> )  ! class 5      |
|      Store Double from BA into Memory Stack                        |
|         CODE ( STD <variable>, <index register> )   ! class 5      |
|      Load G-Relative Address of Variable into A                    |
|         CODE ( LADR <variable> , <index register> ) ! class 5      |
|      Add A to Variable in Memory Stack                             |
|         CODE ( ADM <variable> , <index register> )  ! class 5      |
|      Push Registers to Memory Stack                                |
|         CODE ( PUSH <nnn lll ccc> )                ! class 3       |
|                                                                    |
|                                                              -->   |
```

```
|                                                                    |
|      Pop Memory Stack to Registers                                 |
|         CODE ( POP <nnn lll ccc> )                 ! class 3       |
|                                                                    |
|  Branching                                                         |
|  ---------                                                         |
|                                                                    |
|      Branch if CARRY                                               |
|         CODE ( BIC <label> )                       ! class 2       |
|      Branch unconditionally                                        |
|         CODE ( BUN <label> )                       ! class 2       |
|      Branch on X Less Than A or Increment X                        |
|         CODE ( BOX <label> , <index register> )    ! class 5       |
|      Branch if CC is Greater                                       |
|         CODE ( BGTR <label> )                      ! class 2       |
|      Branch if CC is Equal                                         |
|         CODE ( BEQL <label> )                      ! class 2       |
|      Branch if CC is Greater or Equal                              |
|         CODE ( BGEQ <label> )                      ! class 2       |
|      Branch if CC is Less                                          |
|         CODE ( BLSS <label> )                      ! class 2       |
|      Branch on A Zero                                              |
|         CODE ( BAZ <label> )                       ! class 2       |
|      Branch if CC is not equal                                     |
|         CODE ( BNEQ <label> )                      ! class 2       |
|      Branch on A Not Zero                                          |
|         CODE ( BANZ  <label> )                     ! class 2       |
|      Branch if CC is Less or Equal                                 |
|         CODE ( BLEQ <label> )                      ! class 2       |
|      Branch if no OVERFLOW                                         |
|         CODE ( BNOV <label> )                      ! class 2       |
|      Branch if no CARRY                                            |
|         CODE ( BNOC <label> )                      ! class 2       |
|      Branch Forward Indirect                                       |
|         CODE ( BFI )                               ! class 1       |
|                                                                    |
|  Moves/Compares/Scans                                              |
|  --------------------                                              |
|                                                                    |
|      Move Words                                                    |
|         CODE ( MOVW <m ssd nnn> )                  ! class 3       |
|      Move Bytes                                                    |
|         CODE ( MOVB <m ssd nnn> )                  ! class 3       |
|      Compare Words                                                 |
|         CODE ( COMW <m ssd nnn> )                  ! class 3       |
|      Compare Bytes                                                 |
|         CODE ( COMB <m ssd nnn> )                  ! class 3       |
|      Scan Bytes While                                              |
|         CODE ( SBW <m ssd nnn> )                   ! class 3       |
|      Scan Bytes Until                                              |
|         CODE ( SBU <m ssd nnn> )                   ! class 3       |
|                                                              -->   |
```

```
Program Register Control
------------------------

    Set L with A
       CODE ( SETL )                                  ! class 1
    Set S with A
       CODE ( SETS )                                  ! class 1
    Set E with A
       CODE ( SETE )                                  ! class 1
    Set P with A
       CODE ( SETP )                                  ! class 1
    Read E into A
       CODE ( RDE )                                   ! class 1
    Read P into A
       CODE ( RDP )                                   ! class 1
    Set RP
       CODE ( STRP <register> )                       ! class 4
    Add Immediate Operand to S
       CODE ( ADDS <immediate operand> )              ! class 3
    Set Condition Code to Less
       CODE ( CCL )                                   ! class 1
    Set Condition Code to Equal
       CODE ( CCE )                                   ! class 1
    Set Condition Code to Greater
       CODE ( CCG )                                   ! class 1

Routine Calls/Returns
---------------------

    Procedure CaLL
       CODE ( PCAL <procedure name> )                 ! class 2
    System Call
       CODE ( SCAL <system procedure name> )          ! class 2
    Dynamic Procedure Call
       CODE ( DPCL )                                  ! class 1
    Exit from Procedure
       CODE ( EXIT <number from S> )                  ! class 3
    Debug Exit                    ** Privileged **
       CODE ( DXIT )                                  ! class 1
    Branch to subprocedure
       CODE ( BSUB  <subproc name> )                  ! class 2
    Return from Subroutine
       CODE ( RSUB <number from S> )                  ! class 3

Interrupt
---------

    Reset Interrupt Register      ** Privileged **
       CODE ( RIR )                                   ! class 1
```

-->



```
|                                                                    |
|      Exchange Mask with A          ** Privileged **                |
|         CODE ( XMSK )                                  ! class 1   |
|      Dispatch                      ** Privileged **                |
|         CODE ( DISP )                                  ! class 1   |
|      Interrupt Exit                ** Privileged **                |
|         CODE ( IXIT )                                  ! class 1   |
|                                                                    |
|  Bus                                                               |
|  ---                                                               |
|                                                                    |
|      Send Data over Interprocessor Bus                             |
|         CODE ( SEND )              ** Privileged **    ! class 1   |
|                                                                    |
|  Input/Output                                                      |
|  ------------                                                      |
|                                                                    |
|      Execute Input/Output          ** Privileged **                |
|         CODE ( EIO )                                   ! class 1   |
|      Interrogate Input/Output      ** Privileged **                |
|         CODE ( IIO )                                   ! class 1   |
|      High Priority Interrogate     ** Privileged **                |
|         CODE ( HIIO )                                  ! class 1   |
|                                                                    |
|  Map                                                               |
|  ---                                                               |
|                                                                    |
|      Set Map                       ** Privileged **                |
|         CODE ( SMAP )                                  ! class 1   |
|      Read Map into A               ** Privileged **                |
|         CODE ( RMAP )                                  ! class 1   |
|      Age Map                       ** Privileged **                |
|         CODE ( AMAP )                                  ! class 1   |
|                                                                    |
|  Miscellaneous                                                     |
|  -------------                                                     |
|                                                                    |
|      Read the Switch Register into A                               |
|         CODE ( RSW )                                   ! class 1   |
|      Store A into Switch Register                                  |
|         CODE ( SSW )                                   ! class 1   |
|      No Operation                                                  |
|         CODE ( NOP )                                   ! class 1   |
|      Halt                          ** Privileged **                |
|         CODE ( HALT )                                  ! class 1   |
|                                                                    |
|                                                                    |
|                                                                    |
|                                                                    |
|                                                                    |
|                                                                    |
|                                                                 --> |
```

Decimal Arithmetic Option

Quad Store	
CODE (QST <index register>)	class 4
Quad Load	
CODE (QLD <index register>)	class 4
Quad Add	
CODE (QADD)	class 1
Quad Subtract	
CODE (QSUB)	class 1
Quad Multiply	
CODE (QMPY)	class 1
Quad Divide	
CODE (QDIV)	class 1
Quad Negate	
CODE (QNEG)	class 1
Quad Compare	
CODE (QCMP)	class 1
Convert Quad to Logical (unsigned)	
CODE (CQL)	class 1
Convert Quad to Double	
CODE (CQD)	class 1
Quad Scale Up	
CODE (QUP <scale>)	class 3
Quad Scale Down	
CODE (QDWN <scale>)	class 3
Convert Quad to ASCII	
CODE (CQA)	class 1
Convert ASCII to Quad with Initial Value	
CODE (CAQV)	class 1
Convert ASCII to Quad	
CODE (CAQ)	class 1
Quad Round	
CODE (QRND)	class 1
Convert Quad to Integer (signed)	
CODE (CQI)	class 1
Convert Double to Quad	
CODE (CDQ)	class 1
Convert Integer to Quad	
CODE (CIQ)	class 1
Convert Logical to Quad	
CODE (CLQ)	class 1

PSEUDO OPERATOR CODES

Two pseudo operator codes are included in the set of <mnemonics> recognized by the compiler. They are CON and FULL.

CON is treated as a class 3 instruction. Its function is to emit inline simple or string constants and indirect branch locations.

Some examples:

```
CODE(CON %125);
```

emits an %125 in the next instruction location

```
CODE(CON "the con pseudo operator code");
```

emits 14 words of constant information starting in the next instruction location

```
CODE(CON @labelid);
```

emits a code-relative indirect pointer to "labelid" in the next instruction location

FULL is treated as a class 1 instruction. Its purpose is to specify to the compiler that the register stack is full (i.e., the compiler's internal RP counter is set to seven). No code is emitted for this mnemonic.

CONSIDERATIONS

* The identifier associated with a PCAL or SCAL instruction must be a procedure name

* The identifier associated with a branch instruction must be a label or entry identifier

* The compiler may insert indirect branch cells between instructions emitted in a CODE statement. A branch will be emitted around these areas if required. Normally, these values will be emitted behind the first unconditional branch instruction encountered.

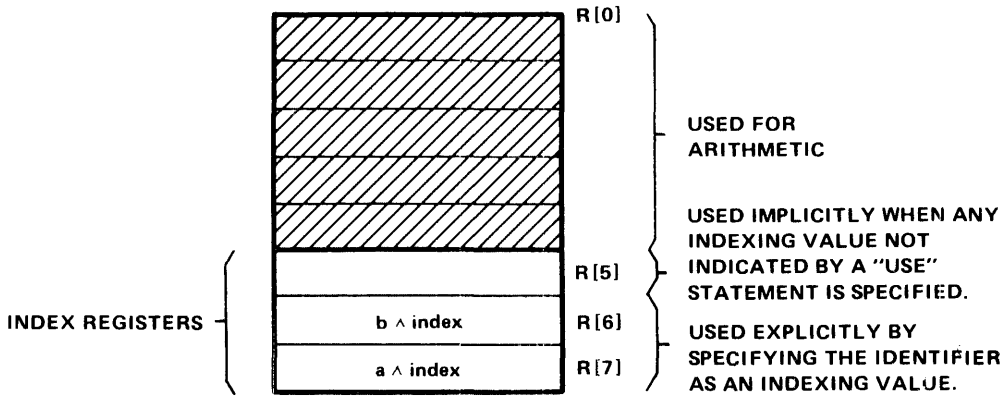
The USE statement assigns an identifier to an index register. This permits the programmer to make explicit references to an index register. The compiler will not use that index register for indexing until the identifier is dropped by a corresponding DROP statement.

The forms of the USE and DROP statement are:

```
USE <name>
--- ------

DROP <name>
---- ------

example

   USE x;
     .
     .
   DROP x;
```

EXAMPLE OF COMPILER INDEX REGISTER ASSIGNMENT

```
USE a^index;
USE b^index;
```



The compiler assigns index registers starting with R[7] down to R[5]

Another example, this time showing a use of an index register.

The programmer knows that the same indexing value will be used throughout a series of statement, therefore and index register is assigned using

```
USE x;
```

Then an initial value is assigned to "x"

```
x := 0;
```

and "x" is used in a loop to access some array elements:

```
DO
  BEGIN
    array[x] := vary;
      .
      .
    n := array[x] + n;
      .
  END
UNTIL (x := x + 1) > 10;
```

The "x" register remains assigned until a DROP statement is encountered:

```
DROP x;
```

A register specified by a USE statement can also be used for temporary storage:

```
PROC p;
  BEGIN
      .
    USE temp;
      .
    temp := n
      .
      .
    RETURN temp;
  END;
```

the index register is dropped when the procedure finishes

CONSIDERATIONS

* The compiler assigns index registers to USE statements starting with R[7].

* An attempt to use more than three index registers will result in an error message being displayed.

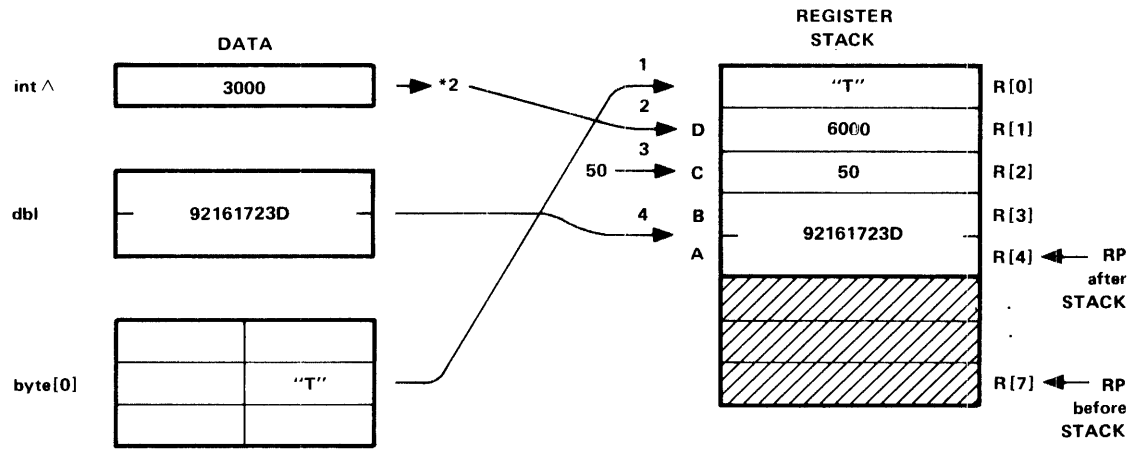
* An error message will be displayed if the compiler needs an index register and none is available.

The STACK statement is used to load a list of variables or values onto the register stack.

The form of the STACK statement is:

```
STACK <expression> , ...
----- ------------

where

   a list of <expressions> is loaded onto the register stack.
   Loading starts at the current setting of RP; RP is incremented
   with each <expression> stacked.  The amount that RP is
   incremented is dependent on the data type of an <expression>.
   The <expressions> are loaded in a left-to-right order

example

   STACK num, 5 * amount, vary;
```

```
EXAMPLE OF STACK STATEMENT
(assume that the register stack is empty)

   STRING .byte[0:3];            ! data declarations.
   INT int^;
   INT(32) dbl;
     .
   STACK byte[3], int^ * 2, 50, dbl;
     .

results in
```



The STORE statement is used to store register stack elements into memory stack variables.

The form of the STORE statement is:

```
STORE <variable> , ... ! maximum of eight <variables>.
----- ----------
```

where

a list consisting of a maximum of eight <variables>, separated by commas, indicates where register stack elements are to be stored. The STORE begins at the current RP setting; RP is decremented with each element stored. The amount RP is decremented is dependent on the data type of the variable where the individual element is stored. Storing occurs to the list of <variables> in a left-to-right order

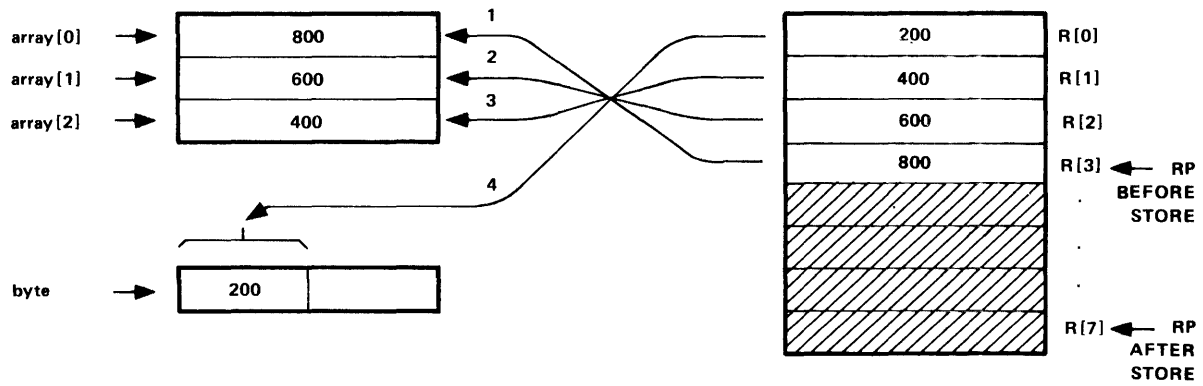
example

```
STORE v1, v2, v3;
```

EXAMPLE OF STORE STATEMENT

```
INT .array[0:2];
STRING byte;
   .
STORE array[0], array[1], array[2], byte;
```

results in



If an index register is assigned before executing a FOR statement, using that index register as the <variable> part in a FOR statement that has a <step> value of one causes the compiler to emit a BOX instruction to implement the FOR loop.

For example:

```
    .
USE index;

FOR index := 0 TO length DO
  BEGIN
      .
      .
    array[index] := n;
  END;
```

causes the compiler to emit a BOX instruction to implement the FOR loop.

Note that the compiler would not emit a BOX instruction in this case if the <step> value of the FOR statement was not one. There is precaution that should be taken in this instance. The register stack should not be modified inside the FOR loop unless it is restored before the end of the loop part.

There is a restriction to the use of the FOR statement that uses the BOX instruction. FOR statements of the form

```
USE x;
FOR x := x TO x + 5 DO ... ;
```

cannot be used. That is, where the <variable> part is also used to compute the <limit> value.

There is another precaution to be taken when using FOR loops. When arithmetic expressions are specified for the <limit> and <step> parts of the FOR statement, the top two memory stack locations are used as temporary storage of these values. Therefore, the top-of-memory-stack area should not be modified in the loop part (unless explicitly restored) and PUSH and POP instructions should not be executed.

These two standard functions provide: 1) the current setting of the complier's RP setting and 2) the current setting of the SWITCH register switches. These standard functions are <primarys> (see "Expressions").

The form of these expressions is:

```
$RP  returns the current setting of the compiler's RP counter
---

$SWITCHES returns the current setting of the SWITCH register
---------

example

   n := $SWITCHES;
```

These two compiler commands are used to 1) set the compiler's internal RP counter and 2) set the compiler's internal S register counter.

The form of these commands is

RP = <register number>

where

<register number> specifies the value RP is to be set to. If seven is specified, the compiler will consider the registers empty

DECS = <sdec value>

where

<sdec value> is an unsigned integer specifying a value to be subtracted from the compiler's S register counter

example

```
?RP = 4
?DECS = 3
```

Note: Following each high level statement (i.e., not CODE, STACK, or STORE constructs) the compiler's internal RP setting is always - 1 (indicating that the register stack is empty).

An example of the use of ?RP:

```
FOR i := 0 TO 4 DO STACK( i ); ?RP = 4
```

In this example, the compiler cannot keep track of the elements loaded into the register stack. However, the programmer knows that five elements have been loaded and therefore the final RP setting should be four.

An example of the use of ?DECS:

```
SUBPROC sp;

  BEGIN
      .
      .
    STACK parm1, parm2, parm3 ; ! load the parameters into the
                                ! register stack.
    CODE( PUSH %722);           ! push the parameters onto the
                                ! top of the memory stack.
    CODE( PCAL proc^name);      ! call the procedure.
?DECS 3
```

In this example, the parameters to a procedure are put onto the memory stack by the programmer specifying a PUSH instruction rather than being put there as a result of a CALL statement. The EXIT instruction of "proc^name" decrements the hardware S register setting by 3 (for the number of parameter words passed). However, the compiler's internal S register setting is unchanged because the compiler is unaware of how many words of parameters were passed. Therefore, ?DECS 3 is used to decrement the compiler's internal S register setting by 3.

Structures provide a method for describing and accessing a set of related data variables such as the fields of a file record. In an inventory control application, for example, a structure can be defined to contain an item number, its unit price, and the quantity on hand:

```
STRUCT .inventory;
  BEGIN
    INT  item^no,
         price,
         on^hand;
  END;
```

Accessing a structure variable requires the use of qualification:

```
inventory.on^hand := inventory.on^hand - num^sold;
```

Structures can also be arrays. Assume, for example, that this inventory contains 50 items:

```
STRUCT .inventory[1:50];
  BEGIN
    INT  item^no,
         price,
         on^hand;
  END;
```

This declaration generates 50 occurrences or copies of the "inventory" structure, or a total of 150 words of storage. The following statement accesses the 31st occurrence of the "inventory" structure:

```
inventory[31].on^hand := inventory[31].on^hand - num^sold;
```

Structures can also be (or contain) multi-dimensional arrays. For example, this inventory might be stored in two warehouses:

```
STRUCT .warehouse[1:2];
  BEGIN
    STRUCT inventory[1:50];
      BEGIN
        INT  item^no,
             price,
             on^hand;
      END;
  END;
```

This declaration generates two occurrences of the "inventory" structure, or a total of 300 words of storage. The following statement accesses the 20th occurrence of the "inventory" structure in "warehouse" number 2:

```
warehouse[2].inventory[20].on^hand
  := warehouse[2].inventory[20].on^hand - num^sold;
```

The compiler treats a structure as a data variable. This allows the program to manipulate individual items in the structure as in the previous examples, or to treat the structure as a single entity:

```
CALL WRITEUPDATE(filenum,warehouse,$LEN(warehouse));
```

Structures can also be specified as a formal parameter in a PROC or SUBPROC declaration. The actual parameter passed to the procedure can be either a structure name or a structure pointer. Structure pointers are described later in this section.

A structure declaration may contain elementary items, substructure declarations, and FILLER declarations:

* An elementary item is any item that has a type declaration (INT I, for example). Such an item is elementary since it cannot be defined further.

* Substructures provide additional structuring within the primary structure. For example, the primary structure "address" may contain the substructure "name." "Name" may contain the elementary items "last," "first," and "middle." This substructure provides useful documentation by identifying the components of "name." It also provides programming convenience by creating additional identifiers for accessing the data.

* FILLER declarations provide a place-holder for data that is not used in the program. For example, when a program manipulates only part of a file record, the unused portions can be declared as FILLER.

The general form of a structure declaration is

```
STRUCT {<structure heading> [;<structure body>]} ,...;
------  -------------------                      -

<structure heading>  gives the structure a name and identifies
                     it as a structure definition, referral,
                     or template.

<structure body>     contains data declarations (optional),
                     substructure declarations (optional),
                     FILLER declarations (optional), and
                     redefinitions (optional).

<structure heading>

  !definition form.

  STRUCT <name> [ "[" <lower bound> : <upper bound> "]" ] ;
  ------ ------                                          -

    <structure body>

or

  !referral form.

  STRUCT <name> ( <referral> )
  ------ ------ - ---------- -
    [ "[" <lower bound> :<upper bound> "]" ] ;
                                             -

or

  !template form.

  STRUCT <name> (*) ;
  ------  ----  --- -

      <structure body>

The general form of <structure body> is

  BEGIN
  -----

    [ <variable declaration(s)>   ;]
              .
              .
                                                        -->
```

```
        [ STRUCT <substructure declaration(s)>   ;]
                  .
                  .
        [ FILLER  [ <constant expression> ] ; ]
                  .
                  .
        [ <redefinition(s)> ;]

                  .
                  .

      END
      ---

    example

       STRUCT address^record;            !structure heading.
        BEGIN
         STRING soc^sec^no[1:11];        !elementary item.
          STRUCT names;                  !substructure heading.
           BEGIN
            STRING last[1:26],           !elementary item.
                   first[1:26],
                   middle[1:26];
           END;
          STRUCT address[1:3];           !substructure heading.
           BEGIN
            STRING address^line[1:20];   !elementary item.
           END;
          STRING zip[1:5];               !elementary item.
        END;
```

<STRUCTURE HEADING>

The structure heading identifies the structure declaration as having the definition, referral, or template form. Also, the heading supplies the structure or substructure with a name, the direct or indirect addressing mode (this applies to structure headings only, not substructures), and specifies its number of occurrences.

<name>

A structure or substructure name may be any valid T/TAL identifier. Optionally, a structure name may be preceded by the indirection symbol.

Note: The addressing mode of a substructure is always the same as that of the structure in which it appears. Therefore, substructure names may not use the indirection symbol.

<lower bound: upper bound>

When used, the optional bounds specifications indicate the number of occurrences of the structure or substructure being defined. In the previous example, the substructure "address" has the bounds specification [1:3]. This indicates that there are three occurrences of "address^line." Therefore, the compiler allocates 60 bytes of storage for the substructure. The following declarations also allocate 60 bytes. However, the second declaration provides better documentation and simpler access.

```
STRUCT address[1:3];
  BEGIN
    STRING address^line[1:20];
  END;
```

```
STRUCT address;
  BEGIN
    STRING street[1:20],
           city[1:20],
           state[1:20];
  END;
```

To access the first character of a city name in the two examples requires the identifiers "address[2].address^line" for the first declaration and "address.city" in the second. These identifiers use qualification, a technique explained later in this section.

Definition Form:

The definition form of a structure both declares the structure and allocates storage for the structure. By contrast, the template form declares a structure, but allocates no space for it. The referral form allocates storage, but has no body of its own. Instead, the referral form allocates storage for a structure with a body identical to a previously declared structure. The previous declaration may have either the definition form or the template form. All the examples given so far in this section have the definition form.

Referral Form:

A structure declaration by referral assigns the new structure a body identical to that of the referral structure. A referral must name the identifier assigned to a previously defined structure. The referral identifier must be enclosed in parentheses. When a structure heading uses the referral format, no body is declared for the structure.

The following structures are equivalent:

```
STRUCT output^line;                    STRUCT address^record(output^line);
  BEGIN
    STRING soc^sec^no[1:11];
    STRUCT names;
      BEGIN
        STRING last[1:26],
               first[1:26],
               middle[1:26];
      END;
    STRUCT address[1:3];
      BEGIN
        STRING address^line[1:20];
      END;
    STRING zip[1:5];
  END;
```

Notice that data items in "output^line" have the same names as data items in "address^record." These names must be qualified when accessed, as explained later in this section.

Two attributes of the referral structure are not carried over to the new structure: addressing mode and bounds specifications.

The addressing mode of the new structure is determined by the presence or absence of the indirection symbol in its name. Thus, the compiler assigns "output^line" to directly addressed memory and ".output^line" to indirectly addressed memory. For ".output^line," the compiler also also builds a one-word INT pointer containing the base address of the structure. The program should not modify this address.

A structure declaration by referral may include its own bounds specifications when multiple occurrences are required. The compiler ignores any bounds specifications declared in the referral structure. When a structure declaration omits bounds specifications, the compiler allocates memory for only one occurrence of the structure regardless of the bounds specifications of the referral structure.

| You cannot declare a substructure by referring to a previously-defined
| structure. Similarly, substructures cannot be used as referral
| structures.

(*) Template Form:

An asterisk in parentheses following a structure name identifies that structure as a template. The compiler allocates no space to a template structure. Therefore, template structures have meaning only when referenced in subsequent structure declarations. Attempting to access a template structure causes an error since no memory is allocated for the template.

Notice that you can declare a structure template as a global without using any global data space. Any number of procedures and subprocedures can then declare subsequent structures with a body identical to the template's by using the referral structure heading format.

Some examples:

```
STRUCT addr^template(*);              !heading for template structure
  BEGIN
    STRING soc^sec^no[1:11];
    STRUCT names;
      BEGIN
        STRING last[1:25],
               first[1:25],           !structure body
               middle[1:25];
      END;
      STRUCT address[1:3];
        BEGIN
          STRING address^line[1:20];
        END;
      STRING zip[1:5];
  END;
```

As mentioned previously, the template structure is simply a definition that is available within your program. It has no meaning until it is referenced in a subsequent STRUCT declaration:

```
STRUCT .address^record(addr^template);
```

This declaration causes the compiler to allocate 154 bytes of memory for the indirectly addressed "address^record." The compiler also allocates a one-word pointer that contains the base address of "address^record." The "address^record" has a structure identical to that of template.

Multiple occurrences of "address^record" are indicated by the use of bounds specifications:

```
STRUCT .address^record(addr^template)[1:2];
```

This declaration causes the compiler to allocate 308 bytes for two occurrences of "address^record," plus a pointer that contains the base address of "address^record."

<STRUCTURE BODY>

The structure body may contain any or all of the following items:

* variable declarations
* substructure declarations
* FILLER declarations
* redefinitions

Variable Declarations

Variable declarations may include any T/TAL variable declarations except for the following:

* Initialization values are not allowed in structures.

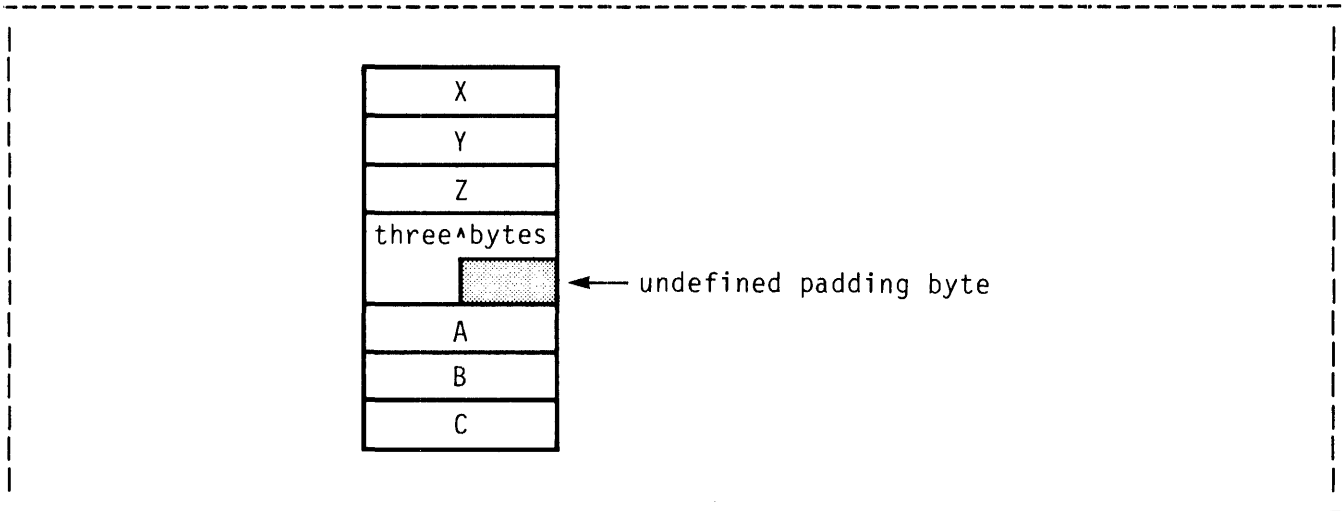
* Bit field declarations are not allowed in structures.

* Structured read-only arrays are not allowed. Such arrays must be initialized, which violates the first rule above.

In certain cases, the compiler adds padding bytes to the variable declarations so that subsequent declarations have the proper word boundary alignment, as in the following example:

```
STRUCT padding^example;
 BEGIN
  INT x,                        !aligned on word boundary
      y,
      z;
  STRING three^bytes[0:2];
                                !<<compiler inserts one byte of
                                !<<padding here
  INT a,
      b,
      c;
 END;
```

The compiler allocates memory for the structure as shown below:



The compiler aligns INT, INT(32), and FIXED items on the next available word boundary, which causes the following conditions:

* The structure may contain padding bytes. For example, a padding is added at the end of the string in the following declarations:

```
INT X;
STRING Y[1:5];
INT Y;
```

* The next available word boundary may not provide maximum efficiency for INT(32) items. INT(32) items are most efficient when aligned on a word boundary with an even address. The compiler generates additional instructions when the item is aligned on a word with an odd address. For maximum efficiency, check the map for this structure in the program listing to determine the INT(32) item's byte offset from the base of the structure. If the offset is an octal number ending in 0 or 4, the item is already aligned at an even word address. Otherwise, you can insert two bytes of FILLER immediately before the INT(32) item declaration to force it to an even word address.

STRUCT <substructure declaration>

A substructure is a structure declaration embedded within another structure or substructure declaration. For example, the previously defined structure "address^record" contains the substructure "names;" "names" contains the elementary items "last," "first," and "middle."

The compiler imposes no practical limitation on the nesting of substructure declarations. (You can declare a substructure within a substructure within a substructure, etc.) This provides all the flexibility required to allow you to define highly complex data structures. Examples of more complex structures are given at the end of this section.

FILLER <constant expression>

FILLER provides a place-holder for data that appears in a data structure, but that is not used in your program. For example, you may be interested in only a few characters of an input record. The unused portions of the record can be defined as FILLER.

The <constant expression> specifies the number of bytes (not words) of FILLER. Bounds specifications are not allowed with FILLER. The expression must evaluate to a positive INT value appropriate for the size of the structure being declared. In many cases, the expression is simply a constant:

```
STRUCT record^part;
 BEGIN
  FILLER 30;
  STRING data[0:29]
  FILLER 10;
 END;
```

You cannot explicitly reference data defined as FILLER since there may

be numerous FILLER items in the same program.

The word FILLER is unique in T/TAL because it is reserved only within the scope of a structure declaration. Again, this is because structures may contain numerous FILLER declarations. Because the word FILLER is reserved only within a structure declaration, you can use it as an identifier elsewhere in your program.

Another use for FILLER is to document the presence of padding bytes used for word boundary alignment within structures, as in the following example:

```
STRUCT padding^example;
  BEGIN
    INT x,
        y,
        z;
    STRING three^bytes[0:2];
    FILLER 1;
    INT a,
        b,
        c;
  END;
```

The FILLER declaration shown above simply documents the presence of a padding byte used for word alignment. The declaration is not required since the compiler provides the padding byte when the declaration is omitted. Such documentation can be helpful when declaring complex structures.

<redefinition(s)>

Substructures and elementary items can be redefined. Redefinition is especially useful when input records have multiple formats. Also, redefinition allows you to assign more than one data type to an item so that it can be handled more efficiently.

The general form for redefinition is:

```
|                                                                    |
|     <identifier> = <elementary item identifier>                    |
|                  -                                                 |
|                                                                    |
|     or                                                             |
|                                                                    |
|     STRUCT <identifier> = <substructure identifier>                |
|                         -                                          |
|     <identifier> can be any valid T/TAL identifier.  The           |
|     indirection symbol (.) cannot precede <identifier> because the |
|                                                                    |
|                                                                    |
|                                                              --> |
```

addressing mode of the redefined item is the same as that of the structure in which it appears.

<elementary item identifier> must be the name assigned to an elementary item previously defined in the current structure declaration.

<substructure identifier> must be the name assigned to a substructure previously defined in the current structure declaration.

example

```
STRUCT sample;
  BEGIN
    STRING letters[1:8];
    FIXED numbers = letters;
  END;
```

The redefinition item and the original item both occupy the same space and have the same offset from the beginning of the structure. The redefinition causes an error in either of two cases:

* The redefined item has different alignment requirements from the original item. This occurs when the original item is a string that begins at an odd byte address, and the redefined item requires word boundary alignment.

* The redefined item is not large enough to contain the original item.

The compiler generates warning messages for these errors.

ACCESSING STRUCTURED DATA

Qualification

Identifiers of data items within structures must be fully qualified when the item is accessed. The general format for qualifying identifiers is:

<struct name>[.<substruct name> ... [.<item name>]]

example

```
address^record.last
```

Full qualification of an identifier depends on the level of nesting used in the structure. You must always precede the identifier with the name of the structure plus the names of any substructures to which it is subordinate. Thus, "item" has different qualification requirements in the following declarations:

```
STRUCT outer;
  BEGIN
    STRUCT inner^1;
      BEGIN
        .
      END;
    STRUCT inner^2;
      BEGIN
        .
      END;
    STRUCT inner^3;
      BEGIN
        INT item;
      END;
  END;
```

```
STRUCT outer;
  BEGIN
    STRUCT inner^1;
      BEGIN
        STRUCT inner^2;
          BEGIN
            STRUCT inner^3;
              BEGIN
                INT item;
                 .
                 .
              ;END;
          END;
      END;
  END;
```

In the first declaration, "item" is subordinate only to "outer" and "inner^3." Thus, "outer.inner^3.item" is the fully qualified identifier for "item." However, in the second example, "item" is subordinate to the structure "outer" and all of its nested substructures. Thus, "outer.inner^1.inner^2.inner^3.item" is the fully qualified identifier for item.

Each identifier in a qualified reference may also have an associated index specification. Therefore, a reference such as "record[I].table[2].item[X]" is possible. The examples later in this section illustrate such references.

Structure Pointer Declaration

The format for declaring pointers for structures is:

```
INT                                                    -->
```

<identifier> is the user-assigned name specified for this pointer.

<referral name> is the identifier assigned to the structure to be accessed through this pointer.

example

```
STRUCT names[1:3];
  BEGIN
    STRING name[1:25];
  END;

  STRING .name^pointer(names);   !String type structure pointer
```

Each structure pointer declaration causes the compiler to allocate one word of storage for an address pointer. The pointer is not initialized; your program must initialize the pointer before using it.

The type declared for a structure pointer must be compatible with the data type to be accessed through the pointer. A type INT pointer can access INT, INT(32), FIXED, and STRING data items. A STRING pointer can only access STRING data items.

EXAMPLES

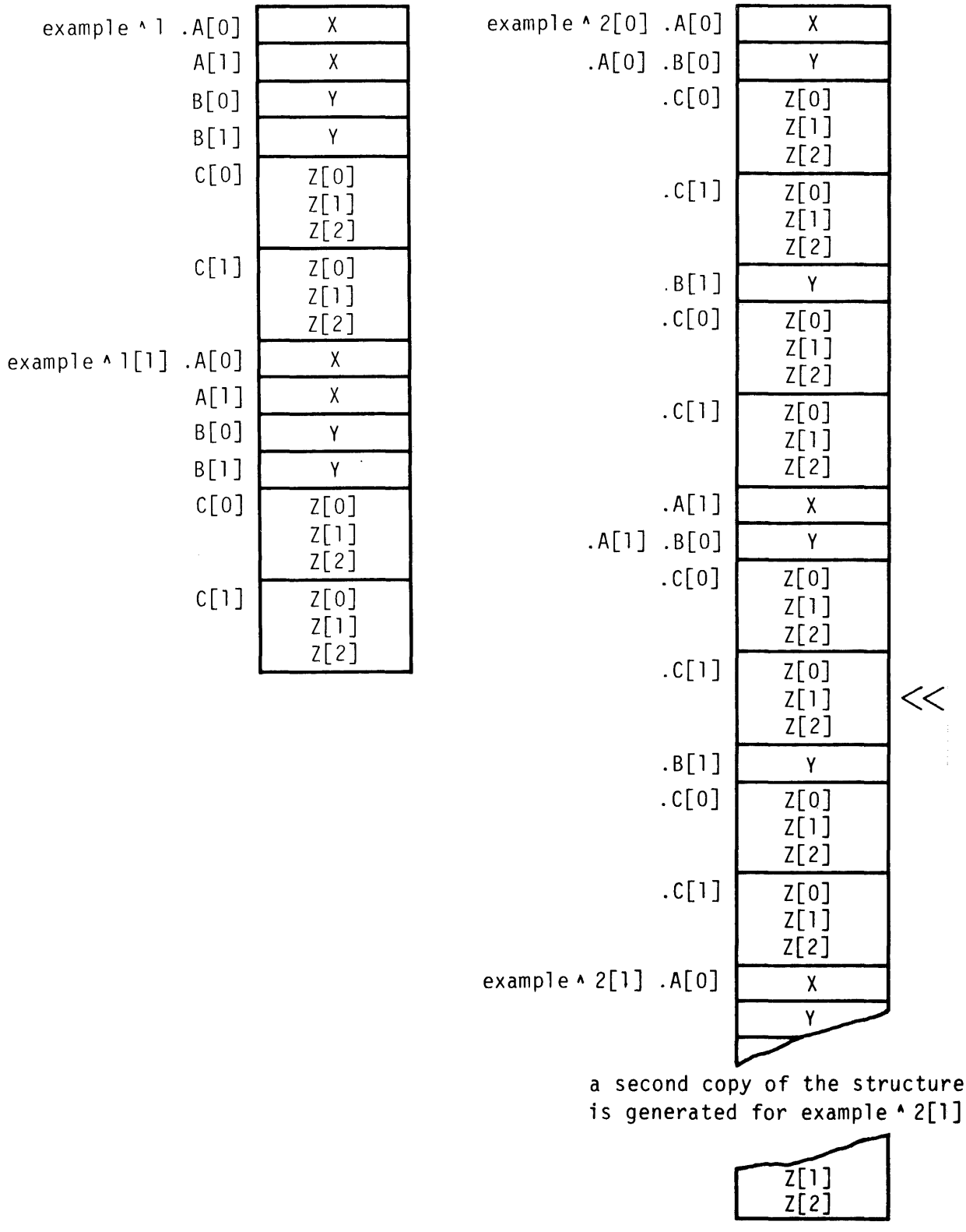
Storage Allocation for Structures

The following examples illustrate the allocation of memory for structure declarations:

```
STRUCT example^1[0:1];
 BEGIN
  STRUCT A[0:1];
   BEGIN
    INT X;
   END;
  STRUCT B[0:1];
   BEGIN
    INT Y;
   END;
  STRUCT C[0:1];
   BEGIN
    INT Z[0:2];
   END;
  END;
```

```
STRUCT example^2[0:1];
 BEGIN
  STRUCT A[0:1];
   BEGIN
    INT X;
     STRUCT B[0:1];
      BEGIN
       INT Y;
        STRUCT C[0:1];
         BEGIN
          INT Z[0:2];
         END;
      END;
   END;
 END;
```

Although examples 1 and 2 both declare four substructures, specify a similar number of occurrences, and use similar identifiers, they are very different structures. The difference results from the different nesting levels in the declarations. Example 1 might be described as a list of arrays; example 2 is a multi- dimensional array (an array containing arrays of arrays). The two are mapped into memory differently and have different qualification requirements. The structures are allocated as follows:


example ^ 1 .A[0]
X
A[1]
X
B[0]
Y
B[1]
Y
C[0]
Z[0]
Z[1]
Z[2]
C[1]
Z[0]
Z[1]
Z[2]
example ^ 1[1] .A[0]
X
A[1]
X
B[0]
Y
B[1]
Y
C[0]
Z[0]
Z[1]
Z[2]
C[1]
Z[0]
Z[1]
Z[2]
example ^ 2[0] .A[0]
X
.A[0] .B[0]
Y
.C[0]
Z[0]
Z[1]
Z[2]
.C[1]
Z[0]
Z[1]
Z[2]
.B[1]
Y
.C[0]
Z[0]
Z[1]
Z[2]
.C[1]
Z[0]
Z[1]
Z[2]
.A[1]
X
.A[1] .B[0]
Y
.C[0]
Z[0]
Z[1]
Z[2]
.C[1]
Z[0]
Z[1]
Z[2]
<<
.B[1]
Y
.C[0]
Z[0]
Z[1]
Z[2]
.C[1]
Z[0]
Z[1]
Z[2]
example ^ 2[1] .A[0]
X
Y
a second copy of the structure
is generated for example ^ 2[1]
Z[1]
Z[2]

Example^2 is clearly the more complex structure. It also has more rigid qualification requirements for access. For example, the fully qualified identifier for the occurrence of Z indicated by the << characters in the illustration is as follows:

```
example^2[0].A[1].B[0].C[1].Z[1]
```

Indexes in this example are given as constants for the sake of clarity. In an application, you are more likely to use variables for indexes.

The amount of qualification required for an identifier depends on how deeply embedded the item is within the structure. For example, example^2.A accesses the first item in the array.

Multi-Dimensional Arrays

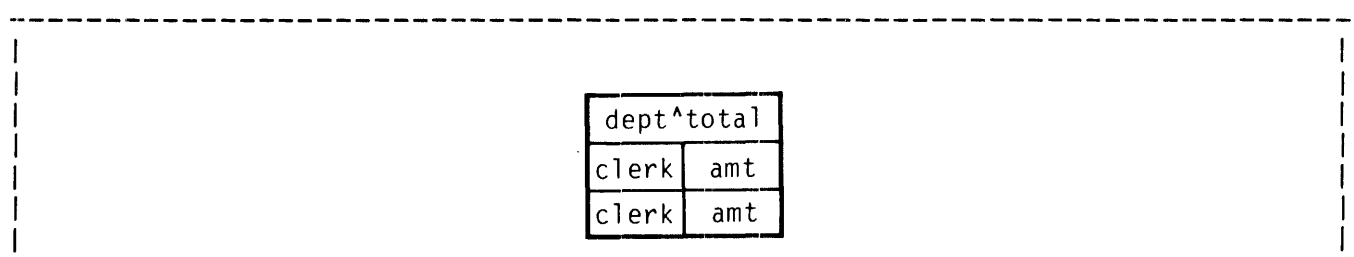
One major use for structures is the definition of file records. Equally important is the structuring of data into multi-dimensional arrays. Multi-dimensional arrays organize data so that it can be manipulated efficiently for a variety of different purposes. The following example, a simple business application, suggests possible uses for multi-dimensional arrays.

Each department in a store has two sales clerks. The following array stores the sales for each clerk and the total sales for the department:

NOTE: The graphic representations in this example illustrate array concepts rather than the actual arrangement of the array in memory. Also, the example is quite simple for the sake of clarity.

```
STRUCT .department;
 BEGIN
  INT(32) dept^total;
  STRUCT indiv^sales[1:2];
   BEGIN
    INT clerk^no,
        amt;
   END;
 END;
```

which is represented as:

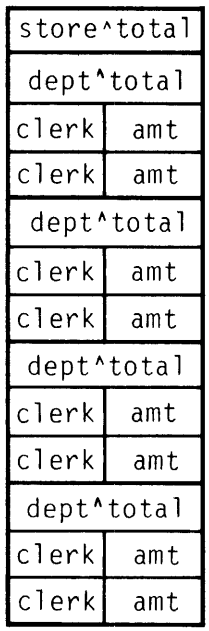


Notice that dept^total can be calculated by adding amt[0] and amt[1].

An array comprising one copy of the previous array for each department contains the store's sales records; the store has four departments:

```
STRUCT .store^sales;
 BEGIN
  INT(32) store^total;
  STRUCT department[1:4];
   BEGIN
    INT(32) dept^total;
    STRUCT indiv^sales[1:2];
     BEGIN
      INT clerk^no,
          amt;
     END;
   END;
 END;
```

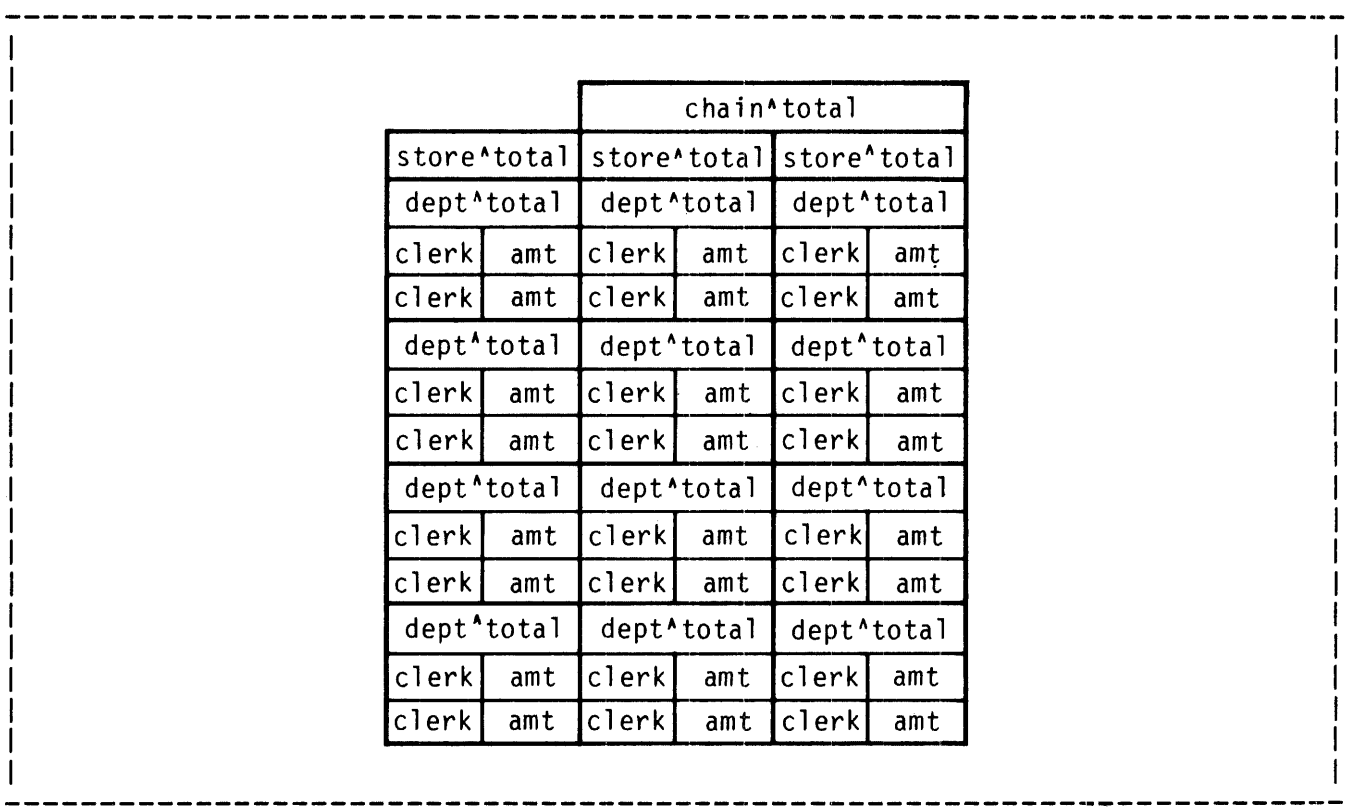
which is represented as:



This store is part of a chain of three stores. An array comprising one copy of the previous array for each store contains the sales records for the chain:

```
STRUCT .chain^sales;
 BEGIN
  FIXED chain^total;
  STRUCT store^sales[1:3];
   BEGIN
    INT(32) store^total;
    STRUCT department[1:4];
     BEGIN
      INT(32) dept^total;
      STRUCT indiv^sales[1:2];
       BEGIN
        INT clerk^no,
            amt;
       END;
     END;
   END;
 END;
```

which is represented as:



The above structure is a three-dimensional array. Subject only to the availability of memory, the structure can expand indefinitely in size and complexity. For example, further structuring can be added to organize data by county, state, region, or perhaps even country.

Notice that the indirection symbol moves to the outermost structure declaration each time a new dimension is added to the array. The substructure declarations may not specify indirection since their addressing mode is the same as that of the array in which they appear.

As mentioned previously, multi-dimensional arrays organize data so that it can be manipulated efficiently for a variety of purposes. Multi-dimensional arrays are efficient because they allow you to create extremely power yet simple FOR, WHILE, and DO loops. Because of its structure, the data can be used many different purposes. For example, the obvious purpose of the relatively simple chain^sales structure is to keep track of total sales by department, store, and for the entire chain. However, the structuring makes it easy to obtain other data such as the following:

* Which product line generates the most (or least) sales revenue for the chain (assuming that corresponding departments sell the same product line throughout the chain).

* Which department generates the most (or least) revenue for a store.

* What are the average sales revenues for a particular department?

In the following examples, S, D, and I are indexes for store^sales, department, and indiv^sales, respectively:

The following statements change individual sales records using a clerk number and amount entered from a terminal:

```
FOR S := 0 TO 2 DO
  FOR D := 0 TO 3 DO
    FOR I := 0 TO 1 DO
      IF chain^sales.store.sales[S].department[D].clark^no[i]
           = entered^clerk^number
      THEN chain^sales.store^sales[S].department[D].amt[I]
           := entered^amount;
```

The following statements update the cumulative department, store, and chain totals. Temp is an INT(32) variable.

```
FOR S := 0 TO 2 DO
  FOR D := 0 TO 3 DO
    BEGIN
      temp := $DBL(chain^sales.store^sales[S].department[D].amt[I]
              + chain^sales.store^sales[S].department[D].amt[I]);
      chain^sales.store^sales[S].dept^total[D]
              := chain^sales.store^sales[S].dept^total[D] + temp;
      chain^sales.store^total[S]
              := chain^sales.store^total[S] + temp;
    END;
chain^total.chain^sales := chain^total.chain^sales
              + store^total[S].chain^sales;
```

Department[2] of this chain sells photographic equipment. The following statements calculate the sales revenues to date for photographic equipment:

```
temp := 0;
FOR S := 0 TO 2 DO
  temp := temp + chain^sales.store^sales[S].dept^total[2];
```

Passing Structures as Parameters

A structure can be specified as a formal parameter in a PROC or SUBPROC declaration. In the following procedure declaration, the "b" and "c" parameters are structures:

```
PROC x(a,b,c);
  INT a;
  STRUCT .b;
    BEGIN
      INT  first,
           last;
      STRING  s1[1:10],
              s2[1:7];
      FILLER 1;
    END,
        .c(chain^sales);
```

In this example, structure c is not related to structure b. This example shows a list of structure definitions; the syntax of the list is the same as the list of integer declarations for "first" and "last. Structure c is a declaration by reference to a previously defined structure.

Additional Examples

The following declarations are used in the remaining examples in this section:

```
STRUCT  .R[0:3];
  BEGIN
  INT    I[0:3};
  STRUCT GROUP1;
    BEGIN
    STRING S1[0:3];
    FILLER 1;
    STRING S2,
           S3[0:5]=S1;
    END;
  STRUCT GROUP2=GROUP1;
    BEGIN
    STRING S1[0:5];
    END;
  END;
INT .P1(R),        !INT structure pointer
    .P2,
     V,
    .TAB[0:$OCCURS(R)*$OCCURS(R.I)*$LEN(R.I)>>1-1];
STRING  .S1(R),    !STRING structure pointer.
        .SP        !STRING pointer.
        .ST[0:5]; !STRING array.

1.    V := R.I;        !store into I[0] from R[0].
```

This is equivalent to the statements

```
      @P1 := @R;       !initialize structure pointer P1.
      V := P1.I;       !store into I[0] from P1.

2.    USE X,Y;
      FOR X := 0 TO 3 DO
       FOR Y := 0 TO 3 DO
        TAB[X<<2+Y]  := R[X].I[Y];
```

This is equivalent to the statements

```
      @P2 := @TAB;
      FOR X := 0 TO 3 DO
       P2  ':=' R[X].I FOR 4*$LEN(R.I)>>1 -> @P2;

 3.    @S1 := @R'<<'1;          !initialize pointer S1 to address of R.
      S1.GROUP1 ':=' "ABCD E"; !fill with string.
```

This is equivalent to the statement

```
       R.GROUP2.S1  ':=' "ABCD E";

4.     @SP := @R[3].GROUP1[2].S1[1];
```

The above statement initializes the STRING pointer SP to point at the specified structure component.

5. `@P2 := @R`

The pointer P2 is initialized to point at the first occurrence of structure R.

6. `ST[5] '=:' R.GROUP2 FOR 6;`

A reverse move of six bytes is performed. A string pointer to the last byte position in GROUP2 is emitted.

7. `ST ':=' R.GROUP2;`

Six bytes (the length of GROUP2) are moved into ST.

STANDARD FUNCTIONS FOR STRUCTURES

The structure functions return information concerning a previously defined data structure. For example, $OFFSET returns the number of bytes from the base of a structure to the beginning of a data item within the structure.

The structure functions always return a constant value. This allows the use of structure functions in LITERAL expressions.

Formats for the structure functions are as follows:

$LEN (<qualified item>)

returns the unit length of an item in bytes. For example, the unit length of a string item is one; the unit length of a substructure is the sum of the lengths of its subordinate items.

$OFFSET (<qualified item>)

returns the number of bytes from the base of the structure to the beginning of a data item within the structure. The base of the structure has offset zero.

$OCCURS (<qualified item>)

returns the number of occurrences of the specified item. For an item with a bounds specification of [0:3], $OCCURS returns the value 4.

$TYPE (<qualified item>)

returns a number that indicates the data type of the specified item. The returned values have the following meaning:

value	data type	value	data type
0	undefined	4	FIXED
1	STRING	5	REAL(32)
2	INT(16)	6	REAL(64)
3	INT(32)	7	Substruct
		8	STRUCT

-->

---

where

<qualified item> is the fully qualified identifier of a structure element. Qualification of identifiers is described in section 2.23 of this manual.

<qualified item> may also be the identifier of a simple variable, but this has little meaning except possibly for the $TYPE function. For simple variables, $LEN returns a unit length, $OFFSET returns zero, and $OCCURS returns 1.

example

```
line ':=' struc^2 FOR $LEN(struc^2) - $LEN(struc^2.sub^7);
```

COMPILER LISTING FOR STRUCTURES

When the MAP option is in effect, structures produce the information shown below.

```
Structure Declaration

  STRUCT SAMPLE;
    BEGIN
      INT X,
          Y,
          Z;
      STRUCT BYTES;
        BEGIN
          STRING CODE[1:2],
                 ITEM[1:7];
          INT    A;
        END;
    END;

Map Listing

   PAGE  n   $VOL.SUBVOL.FILE    [1]                    GLOBAL MAP



   SAMPLE                                VARIABLE,17    STRUCTURE
   1  X                                        0,2     INT
   1  Y                                        2,2     INT
   1  Z                                        4,2     INT
       2  BYTES                               6,7     SUBSTRUCT
       ^    3  CODE                           6,1     STRING
       |    3  ITEM                          10,1     STRING
       |    3  A                             16,2     INT
       |    ^  ^                             ^  ^     ^
        \   |  |                             |  |     |
         \  |  identifier                    |  |     class
          \ |                                |  |     ($TYPE)
           \|                                |  |
            nesting level                    |  |
                                             |  unit length
                                             |  ($LEN value)
                                             |
                                            starting position
                                            ($OFFSET value)
```

* Nesting level is incremented when a substructure is declared within another structure or substructure or when a list of elementary items is encountered.

* Identifier is the identifier specified in the data declaration.

* Starting position is the offset of the item from the base of the structure. The offset is expressed as an octal number.

* Unit length is the length of one occurrence of the data type. Thus, for a structure or substructure unit length is the length of the structure or substructure. Unit length for string data is one; for INT data, 2; for INT(32), 4; and for FIXED data, 10. Unit length is expressed as an octal number.

* Class identifies the data type associated with this identifier.

In addition to the types of data described previously (INT, INT(32), STRING, and FIXED), floating-point data is also available as an option. The <types> associated with floating-point data are:

<type>	Description
REAL	32-bit doubleword
REAL(64)	64-bit quadrupleword

FLOATING-POINT VARIABLES

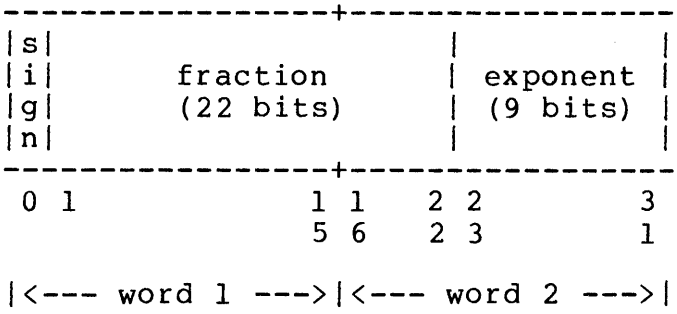
Floating-point variables are defined as <type> REAL. A REAL variable can represent any number in the approximate range

$$\pm 8.62 * 10^{-78} \quad \text{through} \quad \pm 1.16 * 10^{77}$$

Numbers in this range can be represented by a REAL variable with a degree of accuracy approximately equal to seven significant digits.

Internal Format of <type> REAL Data

A REAL variable uses two words of storage. In the following description, visualize the two words as being side by side, with the bits numbered 0 through 31 from left to right.



Bit zero, the leftmost bit of word 1, is the sign bit. If it is 0, it indicates that the number being represented is positive or zero; a 1-bit indicates that the number is negative.

Numbers are represented in "binary scientific notation"; that is, in the form

```
x * 2**y
```

where x is at least 1, but less than 2. For example, the number 4 would be represented as 1 * 2**2, and 10 would be 1.25 * 2**3.

The "x" part of the representation is stored in bits 1 through 22 of the REAL variable. Since x always lies between 1 and 2, its integer part must be 1. Therefore, only the fractional part of x need be stored; the "1" is always assumed.

The "y" part of the representation, the exponent, is stored in bits 23 through 31, the rightmost nine bits of word 2. Using nine bits, exponents from 0 through 511 (octal 777) can be represented. To allow for negative exponents, 256 (octal 400) is added to y before it is stored. Therefore, the exponent is always understood to be the contents of bits <23:31> minus 256. This method provides exponents from -256 (represented by %0) through 255 (represented by %777).

One convention is observed: A bit configuration of all zeros in both words is used to represent the number zero (instead of 1 * 2**-256).

Some examples of REAL representations:

4 = 1.0 * 2**2 represented as 000000 000402

The sign bit is 0, bits <1:22> are 0 (remember that a leading "1" is assumed), and the exponent is %400 + 2, or %402.

-10 = -(1.25 * 2**3) represented as 120000 000403

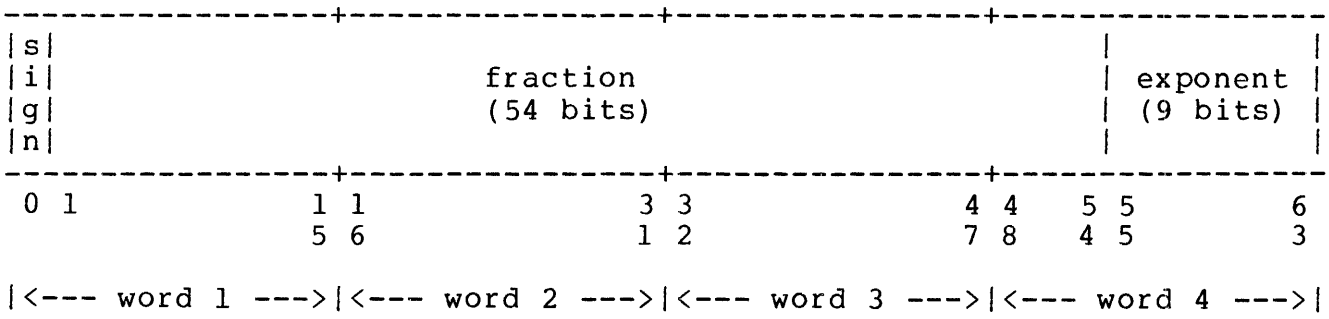
The sign bit is 1. Decimal 1.25 is one and two-eighths, or 1.2 in octal (the 1 is assumed). The exponent is %400 + 3.

EXTENDED FLOATING-POINT VARIABLES

Extended floating-point variables are defined as <type> REAL(64). A REAL(64) variable can represent any number in the same range as that described for REAL variables. The major differnece between the two types is in the degree of precision: A REAL(64) variable is accurate to approximately 17 significant digits.

Internal Format of <type> REAL(64) Data

A REAL(64) variable uses four words of storage. The additional two words provide the increased precision for this type of data.



Bit zero is still the sign bit, and the rightmost nine bits are still the exponent. The fractional part of the representation has been expanded from 22 bits to 54 to provide the additional accuracy.

Example:

```
1/3 = 1.3333... * 2**-2
```

The sign bit is 0, bits <1:54> contain the octal representation of 0.3333... (the 1 is assumed), and the exponent is %400 - 2, or %376. Thus, the internal representation of 1/3 is

```
025252 125252 125252 125376
```

REAL CONSTANTS

Floating-point constants are represented in a T/TAL program with the syntax

[<sign>]<integer part> . <fractional part> E [<sign>]<exponent>

where

<integer part> consists of one or more digits

<fractional part> consists of one or more digits

<exponent> consists of one or two digits

<sign> is "+" or "-"

Examples:

2 can be represented as "+2.0E0," "0.2E+1," "20.0E-1," etc.

-17.2 can be represented as "-17.2E0," "-1720.0E-2," etc.

Note that the integer part, fractional part, "E," and the exponent are required; therefore, 0 must be written as "0.0E0."

REAL(64) CONSTANTS

Extended floating-point constants have the same syntax as REAL constants, except that the letter "L" is used in place of the letter "E." Example:

2 can be represented as "2.0L0," "+0.2L+1," "20.0L-1," etc.

INITIALIZING FLOATING-POINT VARIABLES

Initialization of floating-point variables is performed in the same way as for simple variables, described previously in this manual.

Examples of valid initializations:

```
REAL two^words := 123.456E-43;
REAL(64) four^words := 4211.012739L78;
```

Memory allocation is as described previously:

> REAL values are stored in two consecutive words, REAL(64) values in four.

EQUIVALENCING FLOATING-POINT VARIABLES

Individual words of floating-point quantities can be accessed by equivalencing them to an INT variable. For example, with the declarations

```
REAL(64) four^words;
INT word^one = four^words,
    word^four = word^one + 3;
```

you can access the sign bit and the exponent of "four^words" with the DEFINE statements

```
DEFINE sign^bit = word^one.<0>#,
       exponent = word^four.<7:15>#;
```

The DEFINE statement and bit operations are discussed previously in this manual.

FLOATING-POINT QUANTITIES

Functions of <type> REAL and REAL(64) may be declared, such as

```
REAL PROC real^proc;
```

or

```
REAL(64) PROC real64^proc;
```

These return 2- and 4-word floating-point values, respectively.

Similarly, any procedure may have a REAL or REAL(64) parameter:

```
PROC any^proc( two^words, four^words );
REAL two^words;
REAL(64) .four^words;
```

Note that "four^words" has been declared to be an indirect parameter. This allows "any^proc" to return a value in this variable.

FLOATING-POINT TYPE TRANSFER FUNCTIONS

The following type transfer functions are available for use with floating-point data. Some are existing functions, described previously in this manual, that have been modified to operate with floating-point values; others are expressly designed for floating-point conversions.

$INT (<expression>)

Converts an INT(32), FIXED(0), REAL, or REAL(64) to an INT.

$INTR (<expression>)

Same as $INT, except that a rounding conversion is applied.

$DBL (<expression>)

Converts an INT, FIXED(0), REAL, or REAL(64) to an INT(32).

$DBLR (<expression>)

Same as $DBL, except that a rounding conversion is applied.

$FIX (<expression>)

Converts an INT, INT(32), REAL, or REAL(64) to a FIXED(0).

$FIXR (<expression>)

Same as $FIX, except that a rounding conversion is applied.

$FLT (arg)

Converts an INT, INT(32), FIXED, or REAL(64) to a REAL.

$FLTR (arg)

Same as $FLT, except that a rounding conversion is applied.

$EFLT (arg)

Converts an INT, INT(32), FIXED, or REAL to a REAL(64).

$EFLTR (arg)

Same as $EFLT, except that a rounding conversion is applied.

Conversion Considerations

In conversions from FIXED to REAL or REAL(64), and from REAL or REAL(64) to FIXED, the FIXED quantity is always treated as an integer; that is, as a FIXED(0). For example, the functions

$FLT(1), $FLT(0.1), and $FLT(0.00001)

all yield the same floating-point value. It is the responsibility of the programmer to perform the appropriate scaling.

APPENDICES

Constants

```
"[" <constant> , ... "]"

<repetition factor> * "[" <constant> "]"
```

Simple Variable Declaration

```
<type> { <name> [ := <initialization> ] } , ... ;
```

Array Variable Declaration

```
<type> { [ . ] <name> "[" <lower bound> : <upper bound> "]"
         [ := <initialization> ] } , ... ;
```

Read-Only Array Variable Declaration

```
<type> { <name> [ "[" <lower bound> : <upper bound> "]" ] = 'P'
         := <initialization> } , ... ;
```

Pointer Variable Declaration

```
<type> { . <name> [ := @<variable> "[" <index> "]" ] } , ... ;
```

Dynamic Initialization of Pointer Variable

```
@ <pointer variable> := @<variable> [ "[" <index> "]" ]
```

Pointer Variable Address Conversion

```
<string pointer> := @<word variable> [ "[" <index> "]" ] '<<' 1

<word pointer> := @<string variable> [ "[" <index> "]" ] '>>' 1
```

Equivalencing

```
<type> { [ . ] <name> = <variable> [  <word offset>  ] } , ... ;
```

Base Address Equivalencing

```
                      { 'G'  }
                      { 'L'  }
                      { 'S'  } [ "[" <word index> "]" ]
<type> [ . ] <name> = { 'SG' } [ [+|-] <word offset>  ]
```

LITERAL Declaration

```
LITERAL { <name> = <constant> } , ... ;
```

-->

```
  DEFINE Declarations

DEFINE { <name> = <block of text> # } , ... ;

DEFINE { <name> ( <formal parameter name> , ... )
         = <block of text> # } , ... ;

  Procedure Declaration

  [ <type> ] PROC <name> [ <attributes> ] ;
or
  [ <type> ] PROC <name> ( <formal parameter name> , ... )
    [ <attributes> ] ;
    <parameter specifications>
    BEGIN
      [ local declaration ]
                .
                .
      [ local declaration ]
      [ subprocedure declaration ]
                .
                .
      [ subprocedure declaration ]
      [ [ <statement> ] ; ]
                .
                .
      [ [ <statement> ] ; ]
    END ;
or
    FORWARD ;  or  EXTERNAL ;

  Subprocedure Declaration

  [ <type> ] SUBPROC <name> [ VARIABLE ] ;
or
  [ <type> ] SUBPROC <name> ( <formal parameter name> , ... )
    [ VARIABLE ] ;
    <parameter specifications>
    BEGIN
      [ sublocal declaration ]
                .
                .
      [ sublocal declaration ]
      [ [ <statement> ] ; ]
                .
                .
      [ [ <statement> ] ; ]
    END ;
or
    FORWARD ;
                                                              -->
```

```
  ENTRY Declaration

ENTRY <entry point name> , ... ;

  LABEL Declaration

LABEL <label name> , ... ;

  Bit Extraction

<primary> . "<" <left bit> [ : <right bit> ] ">"

  Bit Deposit

<variable> . "<" <left bit> [ : <right bit> ] ">" := <expression>

  Bit Shift

<primary> <shift op> <number of positions>

  Arithmetic Expression: General Form

[ + | - ] <primary> [ <arith op> <primary> ] ...

  Arithmetic Expression: Assignment Form

<variable> [ . "<" <left bit> ">" [ : "<" <right bit> ">" ] ]
   := <arithmetic expression>

  Arithmetic Expression: IF THEN Form

IF <conditional expression> THEN <arithmetic expression>
                            ELSE <arithmetic expression>

  Arithmetic Expression: CASE Form

CASE <index> OF
  BEGIN
    <expression for index = 0> ;
    <expression for index = 1> ;
       .
       .
       .
    <expression for index = n> ;
    [ OTHERWISE <expression> ; ]
  END

  Conditional Expression

[ NOT ] <condition> [ { AND | OR } [ NOT ] <condition> ] ...
                                                        -->
```

Array Comparison <condition>

```
                        { <s array> FOR <number of elements> }
<d array> <relation> { <constant>                            }
   [ -> <next address> ]
```

<variable> Reference

```
<variable> [ "[" <index> "]" ]
```

Removing Indirection

```
@ <variable> [ "[" <index> "]" ]
```

Specifying Indirection

```
. <direct variable> [ "[" <index> "]" ]
```

Assignment Statement

```
<variable> [ . "<" <left bit> ">" [ : "<" <right bit> ">" ] ]
   := <expression>
```

Compound Statement

```
BEGIN
  [ [ <statement> ] ; ]
            .
            .
            .
  [ [ <statement> ] ; ]
END
```

GOTO Statement

```
GOTO <label>
```

IF Statement

```
IF <conditional expression> THEN [ <statement> ]

IF <conditional expression> THEN [ <statement> ]
ELSE [ <statement> ]
```

-->

CASE Statement

```
CASE <index> OF
  BEGIN
    [ <statement for index = 0> ] ;
    [ <statement for index = 1> ] ;
        .
        .
    [ <statement for index = n> ] ;
    [ OTHERWISE [ <statement> ] ; ]
  END
```

FOR Statement

```
                                { TO     }
FOR <variable> := <initial> { DOWNTO } <limit> [ BY <step> ]
   DO [ <statement> ]
```

WHILE Statement

```
WHILE <conditional expression> DO [ <statement> ]
```

DO Statement

```
DO [ <statement> ] UNTIL <conditional expression>
```

Move Statement

```
          { ':=' } { <s array> FOR <number of elements> }
<d array> { '=:' } { <constant>                         }
   [ -> <next address> ]
```

SCAN Statement

```
{ SCAN  }          { WHILE }
{ RSCAN } <array> { UNTIL } <test character> [ -> <next address> ]
```

CALL Statement

```
CALL <name>

CALL <name> ( { <param 1> , <param 2> , ... <param n> )
```

RETURN Statement

```
RETURN [ <expression> ]
```

CODE Statement

```
CODE ( <instruction> ; ... )
```

-->

USE Statement

USE <name>

DROP Statement

DROP <name>

STACK Statement

STACK <expression> , ...

STORE Statement

STORE <variable> , ... ! maximum of eight <variables>.

Type Transfer Functions

$INT (<dbl expression>)

$HIGH (<dbl expression>)

$DBLL (<int expression> , <int expression>)

$DBL (<int expression>)

$UDBL (<int expression>)

$COMP (<int expression>)

$ABS (<int expression>)

$IFIX (<int expression> , <fpoint>)

$LFIX (<int expression> , <fpoint>)

$DFIX (<dbl expression> , <fpoint>)

$FIXI (<fixed expression>)

$FIXL (<fixed expression>)

$FIXD (<fixed expression>)

Character Test Functions

$ALPHA (<expression>)

$NUMERIC (<expression>)

-->

```
$SPECIAL ( <expression> )

  Min/Max Functions

$MIN ( <arithmetic expression> , <arithmetic expression> )

$MAX ( <arithmetic expression> , <arithmetic expression> )

$LMIN ( <arithmetic expression> , <arithmetic expression> )

$LMAX ( <arithmetic expression> , <arithmetic expression> )

  Carry and Overflow Test Functions

$CARRY

$OVERFLOW

  Fixed Point Functions

$SCALE ( <fixed expression> , <scale> )

$POINT ( <fixed expression> )

  RP Standard Function

$RP

  SWITCHES Standard Function

$SWITCHES

  Compiler Commands

? <compiler command> , ...

  PAGE [ "<heading string>" ]

  listing options:

  -> LIST              NOLIST

  -> MAP               NOMAP

  -> LMAP [ * ]        NOLMAP

  -> CODE              NOCODE

     ICODE          -> NOICODE
                                                              -->
```

```
      INNERLIST       -> NOINNERLIST

      ABSLIST         -> NOABSLIST

   warning control:

   -> WARN               NOWARN

   list only errors control:

      SUPPRESS

   SECTION <section name>

   SOURCE <file name> [ ( <section name> , ... ) ]

   DATAPAGES = <number of pages>

   SETTOG [ 1,2,..., n ]

   RESETTOG [ 1,2,..., n ]

   IF <toggle no.>

   IFNOT <toggle no.>

   ENDIF <toggle no.>

   ASSERTION = <assertion level> , <procedure name>

   fixed point rounding control:

      ROUND           -> NOROUND

   RP = <register number>

   DECS = <sdec value>

   TAL Run Command

 TAL [ / [ IN <source file> ] [, OUT <list file> ] / ]

     [ <object file> ]

   XREF Run Command

 XREF [ / [ IN <source file> ] [, OUT <list file> ] / ]
```

The Backus-Naur Form (BNF) syntax is used to describe the grammar of T/TAL. The grammar consists of a number of "productions", each having the form:

```
<syntactic entity>    ::= <expression>

where

   ::= means "is defined as"

   <expression> is one or more sequence of syntactic terms
   (terminal and non-terminal symbols)
```

For example, the "production" defining the <syntactic entity> <declaration> is

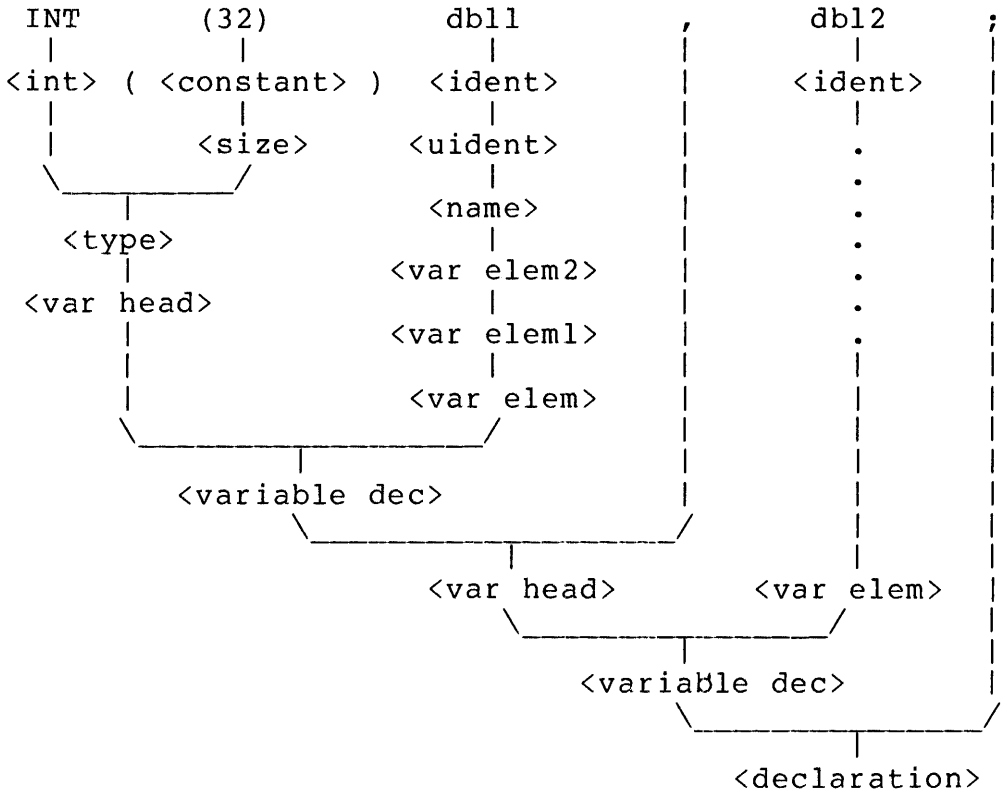
```
<declaration>          ::= <variable dec> ;
                       ::= <routine dec> ;
                       ::= <literal dec> ;
                       ::= <define dec> ;
                       ::= <entry dec> ;
                       ::= <label dec> ;
```

meaning, <declaration> is defined as either a variable declaration, a routine declaration (i.e., PROC or SUBPROC), a literal declaration, a define declaration, an entry declaration, or a label declaration.

The following example shows how a declaration breaks down into its syntactic terms:

```
INT(32) dbl1, dbl2;
```

breaks down as follows:



List of Reserved Symbols

AND	END	LOR	STACK
ASSERT	ENTRY	MAIN	STORE
BEGIN	EXTERNAL	NOT	STRING
BY	FIXED	OF	STRUCT
CALL	FOR	OR	SUBPROC
CALLABLE	FORWARD	OTHERWISE	THEN
CASE	GOTO	PRIV	TO
CODE	IF	PROC	UNTIL
DEFINE	INT	REAL	USE
DO	INTERRUPT	RESIDENT	VARIABLE
DOWNTO	LABEL	RETURN	WHILE
DROP	LAND	RSCAN	XOT
ELSE	LITERAL	SCAN	

List of Terminal Symbols

The Reserved Symbols

!	;	&	
<	(	=	
)	>	*	
@	,	[	
.	]	:	

```
<addop>      ::= + | - | '+' | '-'
<base>       ::= 'G' | 'L' | 'S' | 'SG'
<constant>
<error>
<fclass0>    ::= $CARRY | $OVERFLOW | $RP | $SWITCHES
<fclass1>    ::= $INT | $HIGH | $DBL | $UDBL | $COMP |
                 $ABS | $FIXI | $FIXL | $FIXD | $ALPHA |
                 $NUMERIC | $SPECIAL | $POINT | $PARAM |
                 $LEN | $OFFSET | $OCCURS | $TYPE
<fclass2>    ::= $IFIX | $LFIX | $DFIX | $MIN | $MAX |
                 $LMIN | $LMAX | $SCALE
<gmulop>     ::= * | / | '*' | '/' | '\'
<ident>      ::= <identifier>
<lrelop>     ::= '<' | '=' | '>' | '<=' | '=>' | '<>'
<lshiftop>   ::= '<<' | '>>'
<moveop>     ::= ':=' | '=:'
<opcode>     ::= Tandem 16 Instruction Mnemonic
<replace>    ::= :=  ! assignment !
<saveop>     ::= ->
<text>       ::= <any DEFINE text>
```

List of Productions

```
<program>              ::= <dec head> _:_
                       ::= _:_
<dec head>             ::= <dec group>
<dec group>            ::= <declaration>
                       ::= <dec group> <declaration>
<declaration>          ::= <variable dec> ;
                       ::= <struct dec> ;
                       ::= <routine dec> ;
                       ::= <literal dec> ;
                       ::= <define dec> ;
                       ::= <entry dec> ;
                       ::= <label dec> ;
<entry dec>            ::= <entry head> <entry>
<entry head>           ::= ENTRY
                       ::= <entry dec> ,
<entry>                ::= <uident>
<literal dec>          ::= <literal head> <literal>
<literal head>         ::= LITERAL
                       ::= <literal dec> ,
<literal>              ::= <uident> = <expr>
<uident>               ::= <ident>
<define dec>           ::= <define head> <define>
<define head>          ::= DEFINE
                       ::= <define dec> ,
<define>               ::= <defhead> <text>
<defhead>              ::= <defhead*> =
<defhead*>             ::= <uident>
                       ::= <dparam head> <uident> )
<dparam head>          ::= <uident> (
                       ::= <dparam head> <uident> ,
<label dec>            ::= <label head> <label>
<label head>           ::= LABEL
                       ::= <label dec> ,
<label>                ::= <uident>
<variable dec>         ::= <var head> <var elem>
<var head>             ::= <type>
                       ::= <variable dec> ,
<struct dec>           ::= <struct head> <s-dec body*>
<struct head>          ::= <struct>
                       ::= <struct dec> ,
<s-dec body*>          ::= <s-dec body> <end>
                       ::= <s-referral*>
<struct>               ::= STRUCT
<s-dec body>           ::= <s-dec name*> ; <begin>
                       ::= <s-dec body> <s-item> ;
<s-dec name*>          ::= <s-dec name>
                       ::= <s-dec name> = <adr>
```

-->

List of Productions (cont'd)

```
<s-dec name>          ::= <name>
                      ::= <name> <bounds>
                      ::= <name> ( * )
<s-referral*>         ::= <s-referral>
                      ::= <s-referral> = <adr>
<s-referral>          ::= <name*>
                      ::= <name*> <bounds>
<name*>               ::= <name> ( <ident> )
<s-item>              ::= <s-group>
                      ::= <typed item list>
                      ::= <filler item>
<s-group>             ::= <s-group head> <s-group body> ; <end>
<s-group head>        ::= <struct> <s-group name> ; <begin>
<s-group body>        ::= <s-item>
                      ::= <s-group body> ; <s-item>
<typed item list>     ::= <typed item head> <s-item name>
<typed item head>     ::= <type>
                      ::= <typed item list> ,
<filler item>         ::= FILLER <constant>
<s-group name>        ::= <s-item name>
<s-item name>         ::= <s-item name*>
                      ::= <s-item name*> = <uident>
<s-item name*>        ::= <uident>
                      ::= <uident> <bounds>
<type>                ::= STRING
                      ::= <int> <size>
                      ::= <real> <size>
                      ::= <fixed> <point>
                      ::= <int>
                      ::= <real>
                      ::= <fixed>
<int>                 ::= INT
<real>                ::= REAL
<fixed>               ::= FIXED
<size>                ::= ( <constant> )
<point>               ::= ( <aexpr> )
<var elem>            ::= <var elem1>
                      ::= <var elem1> <replace> <conlist elem>
<var elem1>           ::= <var elem2>
                      ::= <var elem2> = <adr>
<var elem2>           ::= <var elem3>
                      ::= <name> ( <ident> )
<var elem3>           ::= <name>
                      ::= <name> <bounds>
<name>                ::= <uident>
                      ::= . <uident>
<bounds>              ::= [ <expr> : <expr> ]
```

-->

List of Productions (cont'd)

```
<adr>                ::= <ident>
                     ::= <ident> [ <expr> ]
                     ::= <ident> <addop> <primary>
                     ::= <base>
                     ::= <base> [ <expr> ]
                     ::= <base> <addop> <primary>
<conlist elem>       ::= <sum>
                     ::= <conlist head> ]
<conlist head>       ::= <repeat> * <lbrace> <conlist elem>
                     ::= <lbrace> <conlist elem>
                     ::= <conlist head> , <conlist elem>
<repeat>             ::= <primary>
<lbrace>             ::= [
<routine dec>        ::= <spec part> FORWARD
                     ::= <spec part> EXTERNAL
                     ::= <spec part> <body> <statement> <end>
                     ::= <spec part> <body> <end>
<spec part>          ::= <routine type> <spec part*>
<spec part*>         ::= <routine head>
                     ::= <spec part*> <spec> ;
                     ::= <spec part*> <struct spec> ;
<routine head>       ::= <routine name> ;
                     ::= <routine name> <attributes> ;
<routine name>       ::= <uident>
                     ::= <routine head*>
<routine head*>      ::= <rhead> <uident> )
<rhead>              ::= <uident> (
                     ::= <rhead> <uident> ,
<routine type>       ::= PROC
                     ::= <type> PROC
                     ::= SUBPROC
                     ::= <type> SUBPROC
<attributes>         ::= <attribute>
                     ::= <attributes> , <attribute>
<attribute>          ::= CALLABLE
                     ::= PRIV
                     ::= INTERRUPT
                     ::= VARIABLE
                     ::= RESIDENT
                     ::= MAIN
<body>               ::= <local decs>
                     ::= <body> <statement> ;
                     ::= <body> ;
<local decs>         ::= <begin>
                     ::= <local decs> <declaration>
```

-->

List of Productions (cont'd)

```
<s-dec name>          ::= <name>
                      ::= <name> <bounds>
                      ::= <name> ( * )
<s-referral*>         ::= <s-referral>
                      ::= <s-referral> = <adr>
<s-referral>          ::= <name*>
                      ::= <name*> <bounds>
<name*>               ::= <name> ( <ident> )
<s-item>              ::= <s-group>
                      ::= <typed item list>
                      ::= <filler item>
<s-group>             ::= <s-group head> <s-group body> ; <end>
<s-group head>        ::= <struct> <s-group name> ; <begin>
<s-group body>        ::= <s-item>
                      ::= <s-group body> ; <s-item>
<typed item list>     ::= <typed item head> <s-item name>
<typed item head>     ::= <type>
                      ::= <typed item list> ,
<filler item>         ::= FILLER <constant>
<s-group name>        ::= <s-item name>
<s-item name>         ::= <s-item name*>
                      ::= <s-item name*> = <uident>
<s-item name*>        ::= <uident>
                      ::= <uident> <bounds>
<type>                ::= STRING
                      ::= <int> <size>
                      ::= <real> <size>
                      ::= <fixed> <point>
                      ::= <int>
                      ::= <real>
                      ::= <fixed>
<int>                 ::= INT
<real>                ::= REAL
<fixed>               ::= FIXED
<size>                ::= ( <constant> )
<point>               ::= ( <aexpr> )
<var elem>            ::= <var elem1>
                      ::= <var elem1> <replace> <conlist elem>
<var elem1>           ::= <var elem2>
                      ::= <var elem2> = <adr>
<var elem2>           ::= <var elem3>
                      ::= <name> ( <ident> )
<var elem3>           ::= <name>
                      ::= <name> <bounds>
<name>                ::= <uident>
                      ::= . <uident>
<bounds>              ::= [ <expr> : <expr> ]
```

-->

A01 B-5

List of Productions (cont'd)

```
<adr>                 ::= <ident>
                      ::= <ident> [ <expr> ]
                      ::= <ident> <addop> <primary>
                      ::= <base>
                      ::= <base> [ <expr> ]
                      ::= <base> <addop> <primary>
<conlist elem>        ::= <sum>
                      ::= <conlist head> ]
<conlist head>        ::= <repeat> * <lbrace> <conlist elem>
                      ::= <lbrace> <conlist elem>
                      ::= <conlist head> , <conlist elem>
<repeat>              ::= <primary>
<lbrace>              ::= [
<routine dec>         ::= <spec part> FORWARD
                      ::= <spec part> EXTERNAL
                      ::= <spec part> <body> <statement> <end>
                      ::= <spec part> <body> <end>
<spec part>           ::= <routine type> <spec part*>
<spec part*>          ::= <routine head>
                      ::= <spec part*> <spec> ;
                      ::= <spec part*> <struct spec> ;
<routine head>        ::= <routine name> ;
                      ::= <routine name> <attributes> ;
<routine name>        ::= <uident>
                      ::= <routine head*>
<routine head*>       ::= <rhead> <uident> )
<rhead>               ::= <uident> (
                      ::= <rhead> <uident> ,
<routine type>        ::= PROC
                      ::= <type> PROC
                      ::= SUBPROC
                      ::= <type> SUBPROC
<attributes>          ::= <attribute>
                      ::= <attributes> , <attribute>
<attribute>           ::= CALLABLE
                      ::= PRIV
                      ::= INTERRUPT
                      ::= VARIABLE
                      ::= RESIDENT
                      ::= MAIN
<body>                ::= <local decs>
                      ::= <body> <statement> ;
                      ::= <body> ;
<local decs>          ::= <begin>
                      ::= <local decs> <declaration>
```

-->

List of Productions (cont'd)

```
<spec>                ::= <vtype> <ident>
                      ::= <vtype> . <ident>
                      ::= <vtype> <s-pointer>
                      ::= <spec> , <ident>
                      ::= <spec> , . <ident>
                      ::= <spec> , <s-pointer>
<vtype>               ::= <stype>
                      ::= <stype> PROC
                      ::= PROC
                      ::= LABEL
<s-pointer>           ::= . <ident> ( <ident> )
<struct spec>         ::= <struct spec head> <s-spec body*>
<struct spec head>    ::= <struct*>
                      ::= <struct spec> ,
<s-spec body*>        ::= <s-spec body> <end>
                      ::= <s-spec-name> ( <ident> )
<s-spec body>         ::= <s-spec-name> ; <begin>
                      ::= <s-spec body> <s-item> ;
<struct*>             ::= <struct>
<s-spec-name>         ::= . <ident>
                      ::= <ident>
<stype>               ::= STRING
                      ::= INT <size>
                      ::= REAL <size>
                      ::= FIXED <point>
                      ::= FIXED ( * )
                      ::= INT
                      ::= REAL
                      ::= FIXED
<aexpr>               ::= <expr>
                      ::= IF <cexpr> THEN <taexpr> ELSE <aexpr>
                      ::= <case expr head> <case expr body> <end>
                      ::= <case expr head> <case expr body>
                          OTHERWISE <aexpr> ; <end>
                      ::= <left part> <aexpr>
<cexpr>               ::= <aexpr>
<taexpr>              ::= <aexpr>
<case expr head>      ::= <case> <aexpr> OF
<case expr body>      ::= <begin>
                      ::= <case expr body> <aexpr> ;
<expr>                ::= <cond factor>
                      ::= <expr> OR <cond factor>
<cond factor>         ::= <cond secondary>
                      ::= <cond factor> AND <cond secondary>
<cond secondary>      ::= <compare>
                      ::= NOT <compare>
```

-->

List of Productions (cont'd)

```
<compare>              ::= <sum>
                       ::= <relation>
                       ::= <compare head> <compare tail>
<compare head>         ::= <sum> <relation>
<compare tail>         ::= <conlist elem>
                       ::= <conlist elem> FOR <sum>
                       ::= <conlist elem> FOR <sum> <save part>
<sum>                  ::= <term>
                       ::= <addop> <term>
                       ::= <sum> <addop> <term>
                       ::= <sum> <lop> <term>
<term>                 ::= <shift>
                       ::= <term> <mulop> <shift>
<shift>                ::= <primary>
                       ::= <shift> <shiftop> <primary>
<primary>              ::= <constant>
                       ::= <variable>
                       ::= <function designator>
                       ::= ( <aexpr> )
                       ::= <fclass0>
                       ::= <fclass1> ( <aexpr> )
                       ::= <fclass2> ( <faexpr> , <aexpr> )
                       ::= <field designator>
<faexpr>               ::= <aexpr>
<mulop>                ::= *
                       ::= <gmulop>
<lop>                  ::= LOR
                       ::= LAND
                       ::= XOR
<relation>             ::= <
                       ::= =
                       ::= >
                       ::= < =
                       ::= < >
                       ::= > =
                       ::= <lrelop>
<shiftop>              ::= < <
                       ::= > >
                       ::= <lshiftop>
<variable>             ::= <idref>
<idref>                ::= <qual-name>
                       ::= . <qual-name>
                       ::= @ <qual-name>
<qual-name>            ::= <qual-name*>
<qual-name*>           ::= <ident ref>
                       ::= <qual-head> <ident ref>
<qual-head>            ::= <qual-name*> .
<ident ref>            ::= <ident>
                                                          -->
```

List of Productions (cont'd)

```
                          ::= <ident> <open-brkt> <aexpr> ]
  <open-brkt>             ::= [
  <function designator>   ::= <actual head> <actual param> )
                          ::= <actual head> )
  <actual head>           ::= <ident> (
                          ::= <actual head> <actual param> ,
                          ::= <actual head> ,
  <actual param>          ::= <aexpr>
  <label def>             ::= <ident> :
  <statement>             ::= <open stmt>
                          ::= <closed stmt>
  <open stmt>             ::= <if clause> <statement>
                          ::= <if clause>
                          ::= <closure> <open stmt>
                          ::= <for clause> <open stmt>
                          ::= <while clause> <open stmt>
                          ::= <label def> <open stmt>
  <closed stmt>           ::= <do stmt>
                          ::= <go stmt>
                          ::= <case stmt>
                          ::= <assign stmt>
                          ::= <call stmt>
                          ::= <return stmt>
                          ::= <move stmt>
                          ::= <scan stmt>
                          ::= <code stmt>
                          ::= <assert stmt>
                          ::= <use stmt>
                          ::= <drop stmt>
                          ::= <compound stmt>
                          ::= <stack stmt>
                          ::= <store stmt>
                          ::= <closure> <closed stmt>
                          ::= <for clause> <closed stmt>
                          ::= <while clause> <closed stmt>
                          ::= <label def> <closed stmt>
                          ::= <closure>
                          ::= <for clause>
                          ::= <while clause>
                          ::= <label def>
  <if clause>             ::= IF <aexpr> THEN
  <closure>               ::= <if clause> <closed stmt> ELSE
                          ::= <if clause> ELSE
                          ::= <if clause> ; ELSE
  <for clause>            ::= <for head> <to clause> <by clause> DO
                          ::= <for head> <to clause> DO
  <for head>              ::= FOR <forvar> <replace> <aexpr>
  <forvar>                ::= <variable>
                                                                 -->
```

List of Productions (cont'd)

```
<to clause>            ::= TO <aexpr>
                       ::= DOWNTO <aexpr>
<by clause>            ::= BY <aexpr>
<while clause>         ::= WHILE <aexpr> DO
<compound stmt>        ::= <compound head> <end>
<compound head>        ::= <begin> <statement>
                       ::= <begin>
                       ::= <compound head> ; <statement>
                       ::= <compound head> ;
<begin>                ::= BEGIN
<end>                  ::= END
<assign stmt>          ::= <left part> <aexpr>
<left part>            ::= <variable> <replace>
                       ::= <field designator> <replace>
<field designator>     ::= <primary> <range>
<range>                ::= . < <constant> >
                       ::= . < <constant> : <constant> >
<go stmt>              ::= GOTO <ident>
<case stmt>            ::= <case head> <case body> <end>
                       ::= <case head> <case body> <ow-clause>
                           <end>
<ow-clause>            ::= OTHERWISE <statement> ;
                       ::= OTHERWISE ;
<case head>            ::= <case> <aexpr> OF
<case body>            ::= <begin>
                       ::= <case body> <cstmt> ;
                       ::= <case body> ;
<case>                 ::= CASE
<cstmt>                ::= <statement>
<do stmt>              ::= <do> <statement> UNTIL <aexpr>
                       ::= <do> UNTIL <aexpr>
<do>                   ::= DO
<call stmt>            ::= CALL <function designator>
                       ::= CALL <ident>
<return stmt>          ::= RETURN
                       ::= RETURN <aexpr>
<code stmt>            ::= <code head> <instruction> )
<code head>            ::= CODE (
                       ::= <code head> <instruction> ;
<instruction>          ::= <instr>
                       ::= <label def> <instr>
<instr>                ::= <opcode>
                       ::= <opcode> <var part>
                       ::= <opcode> <var part> <xpart>
<var part>             ::= <aexpr>
<xpart>                ::= , <aexpr>
<assert stmt>          ::= <assert head> <aexpr>
<assert head>          ::= ASSERT <constant> :
                                                          -->
```

List of Productions (cont'd)

```
<use stmt>          ::= <use head> <use>
<use head>          ::= USE
                    ::= <use stmt> ,
<use>               ::= <uident>
<drop stmt>         ::= DROP <ident>
                    ::= <drop stmt> , <ident>
<move stmt>         ::= <dest part> <source part> <save part>
                    ::= <dest part> <source part>
<dest part>         ::= <variable> <moveop>
<source part>       ::= <source body>
                    ::= <source part> & <source body>
<source body>       ::= <source var> FOR <sum>
                    ::= <conlist elem>
<source var>        ::= <variable>
<save part>         ::= <saveop> <variable>
<scan stmt>         ::= <scan body> <testword> <save part>
                    ::= <scan body> <testword>
<scan body>         ::= <scan head> UNTIL
                    ::= <scan head> WHILE
<scan head>         ::= SCAN <variable>
                    ::= RSCAN <variable>
<testword>          ::= <sum>
<stack stmt>        ::= STACK <aexpr>
                    ::= <stack stmt> , <aexpr>
<store stmt>        ::= STORE <store part>
                    ::= <store stmt> , <store part>
<store part>        ::= <variable>
                    ::= <field designator>
```

Character	Octal Value (left byte)	Octal Value (right byte)	Meaning
NUL	000000	000000	Null
SOH	000400	000001	Start of heading
STX	001000	000002	Start of text
ETX	001400	000003	End of text
EOT	002000	000004	End of transmission
ENQ	002400	000005	Enquiry
ACK	003000	000006	Acknowledge
BEL	003400	000007	Bell
BS	004000	000010	Backspace
HT	004400	000011	Horizontal tabulation
LF	005000	000012	Line feed
VT	005400	000013	Vertical tabulation
FF	006000	000014	Form feed
CR	006400	000015	Carriage return
SO	007000	000016	Shift out
SI	007400	000017	Shift in
DLE	010000	000020	Data link escape
DC1	010400	000021	Device control 1
DC2	011000	000022	Device control 2
DC3	011400	000023	Device control 3
DC4	012000	000024	Device control 4
NAK	012400	000025	Negatve acknowledge
SYN	013000	000026	Synchronous idle
ETB	013400	000027	End of transmission block
CAN	014000	000030	Cancel
EM	014400	000031	End of medium
SUB	015000	000032	Substitute
ESC	015400	000033	Escape
FS	016000	000034	File separator
GS	016400	000035	Group separator
RS	017000	000036	Record separator
US	017400	000037	Unit separator
SP	020000	000040	Space
!	020400	000041	Exclamation point
"	021000	000042	Quotation mark
#	021400	000043	Number sign
$	022000	000044	Dollar sign
%	022400	000045	Percent sign
&	023000	000046	Ampersand
'	023400	000047	Apostrophe

-->

Character	Octal Value (left byte)	Octal Value (right byte)	Meaning
(	024000	000050	Opening parenthesis
)	024400	000051	Closing parenthesis
*	025000	000052	Asterisk
+	025400	000053	Plus
,	026000	000054	Comma
-	026400	000055	Hyphen (minus)
.	027000	000056	Period (decimal point)
/	027400	000057	Right slant
0	030000	000060	Zero
1	030400	000061	One
2	031000	000062	Two
3	031400	000063	Three
4	032000	000064	Four
5	032400	000065	Five
6	033000	000066	Six
7	033400	000067	Seven
8	034000	000070	Eight
9	034400	000071	Nine
:	035000	000072	Colon
;	035400	000073	Semi-colon
<	036000	000074	Less than
=	036400	000075	Equals
>	037000	000076	Greater than
?	037400	000077	Question mark
@	040000	000100	Commercial at
A	040400	000101	Uppercase A
B	041000	000102	Uppercase B
C	041400	000103	Uppercase C
D	042000	000104	Uppercase D
E	042400	000105	Uppercase E
F	043000	000106	Uppercase F
G	043400	000107	Uppercase G
H	044000	000110	Uppercase H
I	044400	000111	Uppercase I
J	045000	000112	Uppercase J
K	045400	000113	Uppercase K
L	046000	000114	Uppercase L
M	046400	000115	Uppercase M
N	047000	000116	Uppercase N
O	047400	000117	Uppercase O

-->

Character	Octal Value (left byte)	Octal Value (right byte)	Meaning
P	050000	000120	Uppercase P
Q	050400	000121	Uppercase Q
R	051000	000122	Uppercase R
S	051400	000123	Uppercase S
T	052000	000124	Uppercase T
U	052400	000125	Uppercase U
V	053000	000126	Uppercase V
W	053400	000127	Uppercase W
X	054000	000130	Uppercase X
Y	054400	000131	Uppercase Y
Z	055000	000132	Uppercase Z
[	055400	000133	Opening bracket
\	056000	000134	Left slant
]	056400	000135	Closing bracket
^	057000	000136	Circumflex
_	057400	000137	Underscore
`	060000	000140	Grave accent
a	060400	000141	Lowercase a
b	061000	000142	Lowercase b
c	061400	000143	Lowercase c
d	062000	000144	Lowercase d
e	062400	000145	Lowercase e
f	063000	000146	Lowercase f
g	063400	000147	Lowercase g
h	064000	000150	Lowercase h
i	064400	000151	Lowercase i
j	065000	000152	Lowercase j
k	065400	000153	Lowercase k
l	066000	000154	Lowercase l
m	066400	000155	Lowercase m
n	067000	000156	Lowercase n
o	067400	000157	Lowercase o
p	070000	000160	Lowercase p
q	070400	000161	Lowercase q
r	071000	000162	Lowercase r
s	071400	000163	Lowercase s
t	072000	000164	Lowercase t
u	072400	000165	Lowercase u
v	073000	000166	Lowercase v
w	073400	000167	Lowercase w

-->

Character	Octal Value (left byte)	Octal Value (right byte)	Meaning
x	074000	000170	Lowercase x
y	074400	000171	Lowercase y
z	075000	000172	Lowercase z
{	075400	000173	Opening brace
|	076000	000174	Vertical line
}	076400	000175	Closing brace
~	077000	000176	Tilde
DEL	077400	000177	Delete

When the T/TAL compiler detects an error in a source line, it prints a message to inform the programmer of that condition. On the line of the source listing following the erroneous source line, the compiler prints a circumflex symbol (^) to indicate the location of the error within the line. The circumflex is printed under the first character position following the detection of the error (however, because the error detected may involve the relationship of the current source line with a previous line, the circumflex does not always point to the actual error itself). On the next line, the compiler prints a message describing the nature of the error. (Occasionally, a third line may be added to provide supplemental information, such as "IN PROC <procname>," when reference to an earlier procedure is necessary, or "PREVIOUS ON PAGE #<pagenum>," when an error is detected at the bottom of a source listing page and it is necessary to print the diagnostic message on the next page.)

The compiler's diagnostic messages are of two types: error messages and warning messages. The former indicate conditions that must be corrected before the program can be properly compiled. The latter indicate conditions that may or may not be errors, depending on the programmer's intentions and the way in which the program is to be compiled and used; they alert the programmer to recheck his coding to make sure errors are avoided.

Each message begins with a string of four asterisks, the word ERROR or WARNING, an identifying number, and a closing string of asterisks. A brief description of the detected condition then follows. Because these descriptions are necessarily abbreviated, the following text is provided to give a more detailed description of each diagnostic.

NOTES: (1) Because of space limitations on these pages, some messages are continued on a second line. When printed in the compiler listing, however, each message is presented on a single line.
(2) Some diagnostic messages are no longer in use and are not described here. The messages that follow are therefore not in strict ascending numerical sequence.

APPENDIX D: COMPILER DIAGNOSTICS

The following diagnostic messages identify source errors that prevent program compilation or execution, and that must be corrected.

****** ERROR 0 **** COMPILER ERROR <module number>**

A logic error has occurred in the compiler. A 1-digit number identifies the compiler module in which the error was found.

<module number>	
0	= LOAD^OPERAND
1	= AEXPR
2	= INDEX^F
3	= FUN^F
4	= CEXPR
5	= EMIT^CONLIST
6	= BUILDICODE
7	= IDLOOKUP
8	= FIX^BR^IA

Report this occurrence to Tandem Computers Incorporated (please include a copy of the compiler listing).

****** ERROR 1 **** PARAMETER MISMATCH**

The data type of one or more parameters passed to a procedure or subprocedure does not agree with the data types specified for the formal parameters of that routine.

****** ERROR 2 **** IDENTIFIER DECLARED MORE THAN ONCE**

A data declaration contains an identifier that has already been defined within that procedure or subprocedure.

****** ERROR 3 **** RECURSIVE DEFINE INVOCATION**

The <name> in a DEFINE declaration is defined in terms of itself. For example: DEFINE A = B#, B = A#; When A is invoked, it is found to be, in effect, undefined, and the compiler prints the "error 3" message.

****** ERROR 5 **** INT OVERFLOW**

Integer overflow occurred while converting an ASCII number string to binary, or while scaling a quadrupleword constant up or down.

****** ERROR 6 **** ILLEGAL DIGIT**

A number string contains a digit that is greater than the largest number that can be expressed in the specified number base (for example, an octal constant contains the numeral 8).

**** ERROR 7 **** STRING OVERFLOW

A string contains more than 128 characters or extends over more than one line.

**** ERROR 8 **** NOT DEFINED FOR INT(32),FIXED OR REAL

An operation in the current source line produces undefined results when attempted using the stated data types, and is therefore not permitted.

**** ERROR 9 **** ILLEGAL SHIFT COUNT

A bit shift operation specifies an invalid number of positions to be shifted (for example, I << 10000).

**** ERROR 10 **** ADDRESS RANGE VIOLATION

An operation has attempted to specify an address beyond the allowable address limits (for example, INT I = 'G'+300; is invalid since 'G'[255] is the highest address that can be addressed directly).

**** ERROR 11 **** ILLEGAL REFERENCE

A variable reference cannot be used in the present context; a constant is expected.

**** ERROR 12 **** NESTED ROUTINE DECLARATION(S)

One or more PROC declarations have been found within the body of a procedure. A procedure may contain subprocedures only; no other form of nesting is permitted.

**** ERROR 13 **** ONLY INT(16) VALUE(S) ALLOWED

The programmer has attempted to use a STRING, FIXED, or other type of value in an operation that permits only INT values.

**** ERROR 14 **** ONLY INITIALIZATION WITH
CONSTANT VALUE(S) IS ALLOWED

An arithmetic expression can be used to initialize an identifier only if that expression can be evaluated to produce a constant value. Otherwise, only constants can be used for the initialization of variables.

**** ERROR 15 **** INITIALIZATION IS ILLEGAL WITH
REFERENCE SPECIFICATION

The same statement cannot both (1) declare an identifier to be a reference to another identifier and (2) initialize that other identifier (for example, the form INT .A = B := <expression>; is invalid). Separate declarations must be used.

**** ERROR 17 **** FORMAL PARAMETER TYPE SPECIFICATION IS MISSING

If a PROC or SUBPROC declaration contains parameter specifications, the body of the procedure or subprocedure must contain data type declarations for all of the specified parameters. This message indicates that one or more parameters have not been thus defined.

**** ERROR 18 **** ILLEGAL ARRAY BOUNDS SPECIFICATION

The upper and lower bounds specified in an array declaration must be constants or expressions that can be evaluated to produce constants; the value specified for the lower bound must be less than or equal to that of the upper bound. An array declaration in the current source line violates one or more of these requirements.

This message can also result if an equivalenced variable is declared as an array (for example, A[0:5] = B); the bounds specification is ignored.

**** ERROR 19 **** GLOBAL OR NESTED SUBPROC DECLARATION

A SUBPROC declaration has been found that is not within the body of a procedure, or that is contained within a subprocedure. Subprocedures cannot exist independently, nor can one subprocedure contain another.

**** ERROR 20 **** ILLEGAL BIT FIELD DESIGNATOR

In a bit field designator, the ending position number must be greater than or equal to the starting position number, and both must be integer constants. A bit field designator has been found that violates one or both of these requirements.

**** ERROR 21 **** LABEL DECLARED MORE THAN ONCE

Labels must be unique within a given procedure. An identifier followed by a colon, which is identical to a previously-declared label, has been found.

**** ERROR 22 **** BRANCH IDENTIFIER NOT A LABEL

The destination location specified in a GOTO statement must be a label (an identifier followed by a colon). The branch identifier in the current statement has not been declared as a label.

**** ERROR 23 **** VARIABLE SIZE ERROR

Some data type declarations may include size specifications, such as INT(32) or REAL(64); an erroneous size specification has been detected (a declaration of INT(12), for example, is invalid).

**** ERROR 24 **** DATA DECLARATION(S) MUST PRECEDE PROC DECLARATION(S)

A data declaration has been found following the first executable instruction of a procedure or subprocedure.

**** ERROR 26 **** ROUTINE DECLARED FORWARD MORE THAN ONCE

A PROC or SUBPROC declaration followed by the FORWARD declaration should appear only once for any given procedure or subprocedure. The next occurrence of a PROC or SUBPROC declaration containing the same procedure or subprocedure name should be followed by the declarations and statements that make up the actual routine.

**** ERROR 27 **** ILLEGAL SYNTAX

A source entry contains one or more violations of the requirements for its construction.

NOTE: This message can be produced as a result of syntax errors in a source line other than the one actually indicated (it is frequently caused by the absence -- or erroneous presence -- of a semicolon in the preceding line).

**** ERROR 28 **** ILLEGAL USE OF CODE RELATIVE VARIABLE

A code-relative variable (read-only array) cannot be used in the present context.

**** ERROR 30 **** ONLY LABEL OR USE VARIABLE ALLOWED

A DROP operation refers to an identifier that is not a label or a USE variable.

**** ERROR 31 **** ONLY PROC OR SUBPROC IDENTIFIER ALLOWED

In the absence of any ENTRY declarations, a CALL statement can refer only to the name of the procedure or subprocedure that is to be invoked.

**** ERROR 32 **** TYPE INCOMPATABILITY

Certain operations do not permit values of different data types to be used together. For example, an INT(32) value and an INT value cannot be added to each other, and an INT field cannot be moved to a STRING field.

**** ERROR 33 **** ILLEGAL GLOBAL DECLARATION(S)

One or more declarations are invalid in the present context and lexical level (a label, for example, cannot be delared globally).

**** ERROR 34 **** MISSING VARIABLE

A required variable is not present in the current source line.

**** ERROR 36 **** ILLEGAL RANGE

A specified value exceeds the allowable range for a given operation; for example, the statement CODE(QUP 10); is invalid because the maximum range for this instruction is 4.

**** ERROR 37 **** MISSING IDENTIFIER

A required identifier is not present in the current source line.

**** ERROR 38 **** ILLEGAL INDEX-REGISTER SPECIFICATION

Only index registers R[5] through R[7] are available for explicit reference. The programmer has attempted to specify more than three index registers for such use.

**** ERROR 40 **** ONLY ALLOWED WITH A VARIABLE

The FOR <number of elements> option in a comparison or a Move statement can be used only if the item being moved or compared is a variable. For example, IF WORD = "ABCD" FOR 2 is contradictory, since "ABCD" is by definition an entity of four characters.

**** ERROR 42 **** TABLE OVERFLOW <table number>

One of the compiler's tables is completely filled, and an attempt has been made to place another entry in that table. A 1-digit number identifies the particular table.

```
<table number>  0 = CONTAB   (constant table)
                1 = TREE     (expression tree table)
                2 = PLT      (pseudo label table)
                3 = DTEXT    (parametric define table)
                4 = SEC^TAB  (?SOURCE(list) table)
```

No recovery from this condition is possible; changes must be made in the source program to eliminate the overflow problem.

**** ERROR 43 **** ILLEGAL SYMBOL <symbol> or <identifier^name>

The current source line contains a character that is invalid or that is used illegally in the present context.

**** ERROR 44 **** ILLEGAL INSTRUCTION

The specified instruction does not exist in this system.

**** ERROR 45 **** ONLY INT(32) VALUE(S) ALLOWED

The programmer has attempted to use an INT, STRING, or other type of value in an operation that permits only INT(32) values.

**** ERROR 46 **** ILLEGAL INDIRECTION SPECIFICATION

An identifier with the indirection symbol (.) refers to another identifier that has already been declared to be indirect. This implies two levels of indirection; only one level is permitted.

**** ERROR 47 **** ILLEGAL WITH INT(16)

The unsigned divide and unsigned modulo divide operations ('/' and '\' operators) must be of the form

```
INT(32) value  { '/' | '\' }  INT value
```

An INT value has been entered instead of an INT(32) value in the first operand.

**** ERROR 48 **** MISSING <item^specification>

A reference has been made to a label, entry point, procedure, or subprocedure that does not exist. The word MISSING is immediately followed by the identity of the missing item.

**** ERROR 49 **** UNDECLARED IDENTIFIER

A reference has been made to a variable or array that has not been defined in a data type declaration.

**** ERROR 50 **** CAN NOT DROP THIS LABEL

The DROP statement, in addition to deallocating USE registers, can be used to delete labels from the symbol table when they are no longer needed. However, an attempt to DROP a label that has not yet been declared will cause this message to be printed. In a loop, for example, where the label is used in the first iteration only, the programmer should delete it after it is used, rather than at the beginning of (the next iteration of) the loop, since the compiler makes no distinction between iterations.

**** ERROR 51 **** INDEX-REGISTER ALLOCATION FAILED

The compiler was unable to allocate an index register. This can be caused by a condition such as multiple array indexing in a single statement coupled with maximum index register allocation by USE statements.

**** ERROR 52 **** MISSING INITIALIZATION FOR CODE RELATIVE ARRAY

A read-only array must be initialized at the time it is declared.

**** ERROR 53 **** EDIT FILE:INVALID FORMAT OR SEQUENCE <n>

An unrecoverable error has been detected in the source file; <n> is a negative number that identifies the type of error:

-3 = Text file format error
-4 = Sequence error (the line number of the current source line is less than that of the preceding line)

Use the Text Editor program to correct the error.

**** ERROR 54 **** ILLEGAL REFERENCE PARAMETER

A call to a procedure or subprocedure has supplied an indirect (reference) parameter to a formal parameter that has also been declared indirect. This attempts to specify two levels of indirection, when only one is permissible.

**** ERROR 55 **** ILLEGAL SUBPROC ATTRIBUTE

A SUBPROC declaration contains an attribute specification other than VARIABLE, which is the only valid attribute for a subprocedure.

**** ERROR 57 **** SYMBOL TABLE OVERFLOW

The compiler's symbol table is completely filled and an attempt was made to add another entry. Changes in the source program may be required to eliminate this condition, but you may attempt to run the compiler with the MEM run-time option (for example, TAL /IN source, OUT $s, MEM 64/) to increase the number of memory pages available (the default is 48 pages).

**** ERROR 58 **** ILLEGAL BRANCH

If a FOR statement has a USE register as its counter, branching into the FOR loop is not permitted.

**** ERROR 59 **** DIVISION BY ZERO

Division by zero is mathematically undefined and is not permitted.

**** ERROR 60 **** ONLY A DATA VARIABLE MAY BE INDEXED

An index has been appended to an identifier that does not represent a data variable (label, entry point, etc.).

**** ERROR 61 **** ACTUAL/FORMAL PARAMETER COUNT MISMATCH

A call to a procedure or subprocedure has supplied more, or fewer, parameters than were defined in the PROC or SUBPROC declaration.

**** ERROR 62 **** FORWARD/EXTERNAL PARAMETER COUNT MISMATCH

A discrepancy exists between the number of parameters specified in a PROC declaration that is followed by a FORWARD or EXTERNAL declaration and the number specified in the PROC declaration that begins the actual forward or external precedure.

**** ERROR 63 **** ILLEGAL DROP OF USE VARIABLE IN CONTEXT OF FOR LOOP

A DROP statement must not be contained within the body of a FOR loop unless it is preceded, also within the loop, by its corresponding USE statement. If the USE statement is outside the loop and the DROP statement is inside the loop, the USE variable is deallocated with the first iteration; each subsequent execution of the DROP statement attempts to deallocate a variable that is no longer a USE variable.

**** ERROR 64 **** SCALE POINT MUST BE A CONSTANT

The <fpoint> declaration for a FIXED variable, and the <scale> parameter of the $SCALE function, must be an integer constant within the range of -19 through +19. The current source line contains a scale point that is not a constant.

**** ERROR 65 **** ILLEGAL PARAMETER OR ROUTINE NOT VARIABLE

The $PARAM function is used in variable procedures and subprocedures to check for the presence or absence of optional parameters. The <formal parameter> supplied to the $PARAM function is not in the formal parameter list for the routine, or the $PARAM function appears in a routine without the VARIABLE attribute.

**** ERROR 66 **** UNABLE TO PROCESS REMAINING TEXT

This message is usually the result of an extremely poorly structured program, when numerous errors have become compounded and concatenated to the point where the compiler is unable to proceed with the analysis of the remaining source lines.

**** ERROR 67 **** SOURCE COMMANDS NESTED TOO DEEPLY

Source coding invoked by the ?SOURCE command may itself contain a ?SOURCE command to call in other coding, which may, in turn, call still other coding. The maximum limit for such nesting is four levels; that limit has been exceeded.

**** ERROR 68 **** CODE SPACE OVERFLOW

The program has exceeded the amount of memory available for the code area (maximum 64K bytes). The program must be restructured to decrease the size of its coding.

**** ERROR 69 **** INVALID TEMPLATE ACCESS

A template structure has meaning only when referred to in subsequent structure declarations; the compiler allocates no storage space to it. The current source line has attempted to access a template structure as if it were a normal data item.

**** ERROR 70 **** ONLY ITEMS SUBORDINATE TO A STRUCTURE MAY BE QUALIFIED

A qualified reference of the form <name>.<subname>.<itemname> applies only to data items within a structure. The programmer has entered a qualified reference to a data item that is not part of a structure.

**** ERROR 71 **** ONLY INT OR STRING STRUCT POINTERS ARE ALLOWED

The programmer has declared a structure pointer of a data type other than integer or string, the only acceptable types.

**** ERROR 72 **** INDIRECTION MUST BE SUPPLIED

When declaring a structure pointer, the indirection symbol (.) must precede the identifier that is to represent the pointer; the indirection symbol is missing.

**** ERROR 73 **** ONLY STRUCTURE IDENTIFIERS MAY BE USED AS A REFERRAL

In a referral form of a structure declaration, the referral identifier must be a structure identifier, previously declared in either the definition form or the template form of a STRUCT declaration.

**** ERROR 74 **** WORD ADDRESSABLE ITEMS MAY NOT BE ACCESSED THROUGH A STRING STRUCTURE POINTER

Although an INT structure pointer can access items of any data type, a STRING structure pointer can only access STRING data items.

**** ERROR 76 **** ILLEGAL STRUCT OR SUBSTRUCT REFERENCE

A structure or substructure reference may only appear in a MOVE or SCAN statement, or as an address reference. The current source line violates this restriction.

**** ERROR 77 **** STACK SPACE OVERFLOW

The program's data area has become too large for the program to be run (maximum size is 32K, less 512 bytes). Examine the program logic to determine if any data variables can be deleted or arrays decreased in size. (For example, if a number of variables are used in different parts of the program and are never used concurrently, a single "utility" variable could be used instead.)

**** ERROR 78 **** INVALID NUMBER FORM

A floating-point constant has been entered incorrectly. A REAL constant must be written in the form

[<sign>]<integer part> . <fractional part> E [<sign>]<exponent>

and a REAL(64) constant must be entered in the form

[<sign>]<integer part> . <fractional part> L [<sign>]<exponent>

**** ERROR 79 **** REAL UNDERFLOW/OVERFLOW

Underflow or overflow occurred during input conversion of a REAL or REAL(64) number. Floating-point numbers must be within the approximate range

$$\pm 8.62 * 10^{-78} \quad \text{through} \quad \pm 1.16 * 10^{77} .$$

**** ERROR 80 **** INVOKED EXTERNAL PROC CONVERTED TO INTERNAL

The current declaration is attempting to redefine as internal a PROC that has already been called as an external procedure.

**** ERROR 81 **** INVOKED FORWARD PROC CONVERTED TO EXTERNAL

The current declaration is attempting to redefine as external a PROC that has already been called as an internal procedure.

The following diagnostic messages indicate conditions that may or may not affect program compilation or execution. The programmer should recheck his coding carefully to determine whether correction is necessary.

**** WARNING 0 ***** ALL INDEX REGISTERS ARE RESERVED

Three variables have been defined by USE statements as explicit references to index registers. An attempt to reserve another index register will result in an error message.

**** WARNING 1 ***** IDENTIFIER EXCEEDS 31 CHARACTERS IN LENGTH

An identifier in the current source line is longer than the maximum number of characters. All characters following the 31st are ignored.

**** WARNING 2 ***** ILLEGAL OPTION SYNTAX

A compiler control command option has been entered incorrectly; the option is not performed. (This may or may not affect the program itself, depending on the function of the option.)

**** WARNING 3 ***** INITIALIZATION LIST EXCEEDS ALLOCATED SPACE

An initialization list contains more values or characters than can be contained by the variable or array being initialized. The excess items are ignored.

**** WARNING 9 ***** RP MISMATCH

An operation contains conflicting instructions for the use of the register pointer (for example, IF A THEN STACK 1 ELSE STACK 1D;), which cannot be resolved at compilation time.

This message can also occur following a large number of errors that result in an RP conflict.

**** WARNING 10 ***** RP OVERFLOW OR UNDERFLOW

A calculation has produced an index register number that is greater than seven or less than zero.

**** WARNING 12 ***** UNDEFINED OPTION

The programmer has entered a compiler control command option that does not exist. The erroneous command is ignored.

**** WARNING 13 ***** VALUE OUT OF RANGE

A value is in excess of the permissible range for its context (for example, a shift count greater than the number of existing bits).

**** WARNING 14 ***** INDEX WAS TRUNCATED

It is not permissible to equivalence a variable to an indirect variable with an index (for example, INT .S[0:4]; INT S1 = S[1];). The compiler truncates the index from the equivalencing declaration (resulting in INT S1 = S).

**** WARNING 15 ***** RIGHT SHIFT EMITTED

A string address, passed as a parameter when a word address was expected, has been converted to a word address. Truncation of data is possible.

**** WARNING 16 ***** VALUE PASSED AS REFERENCE PARAMETER

A value has been passed to a procedure or subprocedure that expected that parameter to be a reference. If this was the programmer's intent, and if the value can be interpreted as a 16-bit address, no error may be involved.

**** WARNING 17 ***** INITIALIZATION VALUE TOO COMPLEX

An initialization expression is too complicated to evaluate in the current context.

**** WARNING 18 ***** S-REGISTER MISMATCH

A statement contains conflicting instructions for the setting of the S-register. For example, if a subprocedure contains the statement IF A THEN CODE(ADDS 1) ELSE CODE(ADDS 2), the setting of the S-register depends on the evaluation of A, which cannot be resolved at compilation time.

**** WARNING 19 ***** PROC NOT DECLARED FORWARD WITH ABSLIST OPTION ON

If the ABSLIST option is to be used, the compiler must know the size of the PEP table before any actual procedure body is found in the source program. This can be accomplished by either (1) entering a PEP command option at the beginning of the program, or (2) entering a FORWARD declaration for each internal procedure, preceding any procedure having an actual body. This message is printed when the PEP command is not used, and a procedure is encountered for which a FORWARD declaration was not previously entered.

**** WARNING 20 ***** SOURCE LINE TRUNCATED

A source line extends beyond 132 characters. The excess characters are ignored.

**** WARNING 21 ***** ATTRIBUTE MISMATCH

A procedure declaration, followed by the FORWARD declaration, was previously entered with a given set of attributes. When the actual PROC (or SUBPROC) declaration, with coding, was encountered, its attributes did not match those of the previous declaration.

**** WARNING 22 ***** ILLEGAL COMMAND LIST FORMAT

The format of the list of parameters supplied to a compiler control command is incorrect. The command is not performed.

**** WARNING 23 ***** THE LIST LENGTH HAS BEEN USED FOR THE COMPARE COUNT

The FOR <number of elements> specification was omitted from a Compare statement; the actual length of the item to be compared is assumed to be the number of elements intended.

**** WARNING 24 ***** A USE REGISTER HAS BEEN OVERWRITTEN

The evaluation of an expression has caused the value in a USE register to be destroyed (multiplication of two FIXED values, for example, can cause this to occur).

**** WARNING 25 ***** FIXED POINT SCALING MISMATCH

The scale factor of a FIXED value passed as a parameter does not match that of the formal parameter.

**** WARNING 26 ***** MAIN PROCEDURE IS MISSING

In a program that is to be executed, at least one procedure must have the attribute MAIN. If the current source coding is not to be executed but is intended for library use, this message can be ignored.

**** WARNING 27 ***** FPOINT>ABS(19)

The <fpoint> in a FIXED declaration or the <scale> parameter of the $SCALE function is less than -19 or greater than +19. The fixed-point specification is set to its maximum limit, either -19 or +19.

**** WARNING 28 ***** MORE THAN ONE MAIN SPECIFIED. LAST MAIN IS <procedure name>

Although more than one procedure may have the attribute MAIN, only the last such procedure in the program is considered to be the actual main procedure.

**** WARNING 29 ***** ILLEGAL SUBPROC ATTRIBUTE(S)

The only attribute permitted for a subprocedure is VARIABLE. A SUBPROC declaration with additional attributes has been found. The VARIABLE attribute is implemented, but the others are ignored.

**** WARNING 30 ***** PROC FORCED TO 32K BOUNDARY

A procedure cannot extend across the boundary between the upper and lower 32K segments of the code area. If a procedure that would otherwise begin in the lower segment is too large to fit completely within that segment, the compiler automatically maps that procedure into the upper segment, beginning at the boundary. This can cause wasted space at the upper end of the lower segment. If efficiency in memory allocation is important, the sequence of procedures in the source program can be reorganized, or the FORWARD declaration used, to eliminate this condition.

**** WARNING 31 ***** MISSING FOR PART

The FOR <number of elements> specification was omitted from a Move statement. The actual length of the item to be moved is assumed to be the number of elements intended.

**** WARNING 32 ***** RETURN NOT ENCOUNTERED IN TYPED PROC OR SUBPROC

Although a procedure or subprocedure automatically returns control to the calling routine when the last END statement is reached, a function procedure or function subprocedure (identified by a <type> specification in its PROC or SUBPROC declaration) is expected to return a value. To do so, it must contain at least one RETURN statement with an identifier.

**** WARNING 33 ***** REDEFINITION SIZE CONFLICT

When redefining substructures or structure elements, the redefined item must be of sufficient size to contain the original item.

**** WARNING 34 ***** REDEFINITION OFFSET CONFLICT

In the redefinition of a structure element or substructure, the original item is a string beginning at an odd byte address, but the redefined item requires word boundary alignment.

**** WARNING 35 ***** REAL OPERATIONS NOT SUPPORTED

A source line specifies operations involving floating-point values, but the floating-point option has not been installed on this system.

**** WARNING 36 ***** DOUBLE * OR / NOT SUPPORTED

Multiplication and division of INT(32) values is not available.

Page numbers in this index marked by an asterisk (*) indicate primary references for the related subject matter.

READER'S COMMENTS

Tandem welcomes your feedback on the quality and usefulness of its publications. Please indicate a specific *section* and *page* number when commenting on any manual. Does this manual have the desired completeness and flow of organization? Are the examples clear and useful? Is it easily understood? Does it have obvious errors? Are helpful additions needed?

Title of manual(s): _____

FOLD ➤

FOLD ➤

FROM:

Name _____

Company _____

Address _____

City/State _____ Zip _____

A written response is requested, yes no ?

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 482 CUPERTINO, CA. U.S.A.

POSTAGE WILL BE PAID BY ADDRESSEE

TANDEM
COMPUTERS, INC.

Attn: Technical Publications
19333 Vallco Parkway
Cupertino, CA, U.S.A. 95014

FOLD

STAPLE HERE